SIAM J. COMPUT.
Vol. 20, No. 1, pp. 1-21, February 1991

© 1991 Society for Industrial and Applied Mathematics
001

ABSOLUTE FACTORIZATION OF POLYNOMIALS: A GEOMETRIC APPROACH*

DOMINIQUE DUVAL†

Abstract. In this paper a new algorithm is presented for factoring bivariate polynomials over algebraically closed fields. Or, equivalently, for determining the irreducible components of a plane curve. This algorithm is based on properties of some geometric invariants of the curve, and is similar to Berlekamp's algorithm for factorization of univariate polynomials over finite fields.

Key words. computer algebra, polynomial factorization, reducible curves, function fields, divisors, algebraic extensions

AMS(MOS) subject classifications. 68Q40, 12D05, 14H05, 14C20, 12F05

Introduction. In this paper we describe a new algorithm for the determination of the irreducible components of a plane curve. The algorithm has the property that it first determines the number of those components. The main ideas of this algorithm have been presented in [Du1], but here is given a more efficient way to determine the components, once their number is known.

The notions of curve and of irreducible curve here are the algebraic ones: let $\mathbf{K}$ denote an algebraically closed field of characteristic zero (this assumption may be weakened as explained at the end of § 1), for example the field $\mathbf{C}$ of complex numbers. A (*plane algebraic*) *curve* C (over $\mathbf{K}$) is the set of points (x, y) in the affine plane $\mathbf{A}_2(\mathbf{K}) = \mathbf{K} \times \mathbf{K}$ which satisfy $F(x, y) = 0$ for a given nonconstant polynomial $F(X, Y)$ in $\mathbf{K}[X, Y]$. The curve C is *irreducible* if the polynomial $F(X, Y)$ is irreducible in the ring $\mathbf{K}[X, Y]$, i.e., if F is different from $F_1 \times F_2$ for any nonconstant polynomials F_1 and F_2 in $\mathbf{K}[X, Y]$.

The ring $\mathbf{K}(X, Y)$ is *factorial*, which means that every polynomial F in $\mathbf{K}[X, Y]$ has a (essentially) unique *irreducible decomposition*

$$F = \prod_{i=1}^{n} F_i^{k_i}.$$

In this decomposition, each F_i is irreducible in $\mathbf{K}[X, Y]$, F_i is not collinear to F_j if $i \neq j$, and the k_i's are positive integers.

The *irreducible components* of the curve C are the curves C_i of equation $F_i(x, y) = 0$, and k_i is the *multiplicity* of C_i in C. We shall see in § 1 that, thanks to the classical methods of "square-free decomposition," it is always possible to assume that every k_i is equal to one. It follows that the *geometric* question of determining the irreducible components of a curve is equivalent to the *algebraic* question of **factoring a bivariate polynomial over an algebraically closed field**.

Absolute factorization should be distinguished from *rational factorization*, by which we mean factorization of univariate or multivariate polynomials over a given nonalgebraically closed field. Rational factorization has been much more intensively studied and implemented than absolute factorization. One of the conclusions of this paper is that both kinds of factorization are independent. This can also be deduced from

* Received by the editors February 16, 1988; accepted for publication (in revised form) April 26, 1989.

† Faculté des Sciences, Laboratoire de Théorie des Nombres et Algorithmique, F-87060 Limoges Cedex, France. This work was partially supported by the Centre National de la Recherche Scientifique through Greco de Calcul Formel and PRC Mathématiques-Informatique.

Kaltofen's algorithm (cf. § 1). But this fact has not always been clear, partly because rational factorization used to be considered essential to represent algebraic numbers.

Our algorithm relies on the knowledge of a basis for some vector spaces related to the curve C. These spaces are used in numerous different applications. We are not aware of any easy algorithm to compute them, though different methods have been known since the end of the 19th century. In this paper we give a new algorithm based on one of those methods.

We also compare our algorithm with other algorithms for absolute factorization, using a similar frame for their description.

In § 1 we begin with some general considerations, especially about computations in algebraically closed fields (referring to our work with Dicrescenzo). We then describe within a similar frame three "algebraic" methods for absolute factorization of bivariate polynomials: the methods by Trager and Traverso, by Kaltofen, and ours. In the description of our algorithm appear some finite-dimensional $\mathbf{K}$-vector spaces denoted $L(0)$ and $L(-\wp)$. The rest of the paper is devoted to the definition of these vector spaces, their computation, and the proof of our algorithm.

Section 2 introduces some geometric objects that will be useful, mainly, the notion of a divisor D on a curve C, and the $\mathbf{K}$-vector space $L(D)$ of functions on C which is classically associated to D. Nothing is new in this section, since the curve C is assumed to be irreducible.

In § 3, similar objects are defined when the curve C is reducible. The main result of the paper is proved: *The number of irreducible components of C is equal to the dimension of the $\mathbf{K}$-vector space $L(0)$.*

A way of recovering an irreducible component of C from some $\mathbf{K}$-vector space $L(-\wp)$ is then described in § 4.

Finally, in § 5 a method is described for the computation of bases of the spaces $L(0)$ and $L(-\wp)$. This method dates back to Dedekind and Weber, but here we give a new "rational" version of it, with emphasis on the treatment of algebraic numbers.

1. Three algorithms. In this section are described without proofs three different methods for absolute factorization of bivariate polynomials. They have in common their algebraic flavor, and their strategy. They determine a simple point (α, β) on the given curve C and then compute the irreducible component of C through this point, or equivalently the irreducible factor $\Phi(X, Y)$ of $F(X, Y)$ in $\mathbf{K}[X, Y]$ such that $\Phi(\alpha, \beta) = 0$. Algorithm I is described by Trager [Trag2] and Traverso [Trav], Algorithm II is due to Kaltofen [Ka], and Algorithm III is the one detailed in this paper. Let us also mention the absolute irreducibility test by Heintz and Sieveking [HS] which needs many simple points on C, and the topological method by Bajaj et al. [BCGW] which is restricted to $\mathbf{K} = \mathbf{C}$.

The field K. The field $\mathbf{K}$ is algebraically closed and has characteristic zero. Since we want to perform computations on it, we have to be somewhat more precise. Let us call the four operations $+$, $-$, $\times$, and $/$ the *field operations* on $\mathbf{K}$.

Definition. Here a field $\mathbf{L}$ is a *computable field* if we can represent its elements in such a way that equality can be tested, and if we have for each field operation $*$ on $\mathbf{L}$ an algorithm with input representatives of two elements a and b of $\mathbf{L}$ (with $b \neq 0$ in the case of division) and with output a representative of the element $a * b$ of $\mathbf{L}$.

Example. It is well known that the field $\mathbf{Q}$ of rational numbers is a computable field. It is also well known that if $\mathbf{L}$ is a computable field, if $P(T)$ is an irreducible polynomial with coefficients in $\mathbf{L}$, and if β is a root of $P(T)$ in an algebraic closure $\bar{\mathbf{L}}$ of $\mathbf{L}$, then the field $\mathbf{L}(\beta)$ generated by β and $\mathbf{L}$ in $\bar{\mathbf{L}}$ is a computable field. The reason

is that $\mathbf{L}(\beta)$ is then isomorphic to the quotient $\mathbf{L}[T]/(P(T))$. Note that $P(T)$ usually has several different roots β in $\bar{\mathbf{L}}$, but thanks to the Galois theory we know that they all behave in the same way. This point may be stated as:

"Let β be *any root of* $P(T)$ in $\bar{\mathbf{L}}$, then $\mathbf{L}(\beta)$ is a computable field"

which emphasizes the fact that usually in applications $P(T)$ is given and the following instruction is needed:

"Let β be any root of $P(T)$ in $\bar{\mathbf{L}}$."

But usually in applications this instruction is needed for some nonconstant polynomial $P(T)$ in $\mathbf{L}[T]$, which may be reducible in $\mathbf{L}[T]$. Of course *if* we are given a factorization algorithm on $\mathbf{L}[T]$ (which is *not* a consequence of the fact that $\mathbf{L}$ is a computable field), then $\mathbf{L}(\beta)$ is a computable field, isomorphic to some quotient $\mathbf{L}[T]/(Q(T))$ for an irreducible factor $Q(T)$ of $P(T)$ in $\mathbf{L}[T]$. More precisely, consider some computation over $\mathbf{L}(\beta)$ involving equality tests and field operations. This computation may be considered as the evaluation of a function $\varphi: A \to B$ at some $a \in A$. Let $Q_i(T)$ (for $1 \leqq i \leqq I$) denote the distinct irreducible factors of $P(T)$ in $\mathbf{L}[T]$. For each i, let $b_i \in B$ denote the value of $\varphi(a)$ for any root β of $Q_i(T)$. The result of the evaluation of $\varphi(a)$ may then be represented as the finite set of pairs $\{(b_1, Q_1(T)), (b_2, Q_2(T)), \cdots, (b_I, Q_I(T))\}$.

In this paper we do not use any factorization algorithm in $\mathbf{L}[T]$, but rather we use *dynamic evaluation* [DD]. Consider some computation as above, considered as the evaluation of a function $\varphi: A \to B$ at some $a \in A$. With dynamic evaluation, the result of the evaluation of $\varphi(a)$ is a finite set $\{(b_1, P_1(T))\ (b_2, P_2(T)), \cdots, (b_J, P_J(T))\}$ of pairs with $b_j \in B$ and $P_j(T)$ a factor of $P(T)$ in $\mathbf{L}[T]$. For each j, the value of $\varphi(a)$ is equal to b_j for any root β of $P_j(T)$, and the $P_j(T)$'s form a *splitting* of $P(T)$. Generally, a splitting of $P(T)$ is defined as a set $\{P_j(T)\}_j$ of nonconstant polynomials in $\mathbf{L}[T]$ such that every root of $P(T)$ in $\bar{\mathbf{L}}$ is a root of one and only one $P_j(T)$, and conversely every root of some $P_j(T)$ in $\bar{\mathbf{L}}$ is a root of $P(T)$. Since here we are interested in fields of characteristic zero we give a more restrictive definition, which will simplify the description of the algorithms below. We require that every $P_j(T)$ be square-free, so that a splitting of $P(T) \in \mathbf{L}[T]$ can be defined as a set $\{P_j(T)\}_{1 \leqq j \leqq J}$ of nonconstant polynomials in $\mathbf{L}[T]$ such that $\tilde{P}(T) = P_1(T)P_2(T) \cdots P_J(T)$ where $\tilde{P}(T) = P(T)/\gcd(P(T), P'(T)) \in \mathbf{L}[T]$ is the square-free polynomial associated to $P(T)$. Of course the set of distinct irreducible factors of $P(T)$ in $\mathbf{L}[T]$ form a splitting of $P(T)$, but usually a given computation only leads to a partial factorization of $\tilde{P}(T)$. The cost (in term of elementary operations in $\mathbf{L}$) of each elementary operation in $\mathbf{L}(\beta)$ is similar to the cost if $P(T)$ were irreducible in $\mathbf{L}[T]$, except for equality tests which now have a cost similar to divisions, since they need a gcd computation with $\tilde{P}(T)$ in $\mathbf{L}[T]$. In particular, this cost is independent of the number J of *branches* in the splitting.

Example. In some examples below we consider the polynomial $P(T) = T^2 - q$ over $\mathbf{Q}$ for an arbitrary rational number $q \neq 0$. Then $P(T)$ may be irreducible in $\mathbf{Q}[T]$ or not, i.e., the ring $\mathbf{Q}[T]/(T^2 - q)$ may be a field (then equal to $\mathbf{Q}(\beta)$) or not, depending on the value of q. For example $P(T)$ is reducible in $\mathbf{Q}[T]$ if $q = 1$, and irreducible if $q = 2$. However, computations in the field $\mathbf{Q}(\beta)$ will run the same way for every q.

DEFINITION. An algebraically closed field $\bar{\mathbf{L}}$ is a *computable algebraically closed field* if it is a computable field and if we are able, given any nonconstant univariate polynomial $P(T)$ with coefficients in $\bar{\mathbf{L}}$, to represent any root of $P(T)$ in $\bar{\mathbf{L}}$. We may then use instructions like:

"Let β be any root of $P(T)$ in $\bar{\mathbf{L}}$."

Example. It is a consequence of dynamic evaluation that the field $\bar{\mathbf{Q}}$ of algebraic numbers is a computable algebraically closed field.

More generally, dynamic evaluation proves that if a field $\mathbf{L}$ is a computable field, then its algebraic closure $\bar{\mathbf{L}}$ is a computable algebraically closed field. It may be proved recursively on the number of instructions like "let β be any root of $P(T)$ in $\bar{\mathbf{L}}$" which are used. The main point in this recursion has been considered above: If $P(T)$ is a nonconstant polynomial with coefficients in $\mathbf{L}$ and if $\beta \in \bar{\mathbf{L}}$ is any root of $P(T)$, then the field $\mathbf{L}(\beta)$ generated by β and $\mathbf{L}$ in $\bar{\mathbf{L}}$ is a computable field.

Our *computability assumption* is that there is some computable subfield $\mathbf{K}_0$ of $\mathbf{K}$ which contains all the coefficients of $F(X, Y)$. Every computation will occur in the algebraic closure $\bar{\mathbf{K}}_0$ of $\mathbf{K}_0$, which is a computable algebraically closed subfield of $\mathbf{K}$, so that we may also assume that $\mathbf{K} = \bar{\mathbf{K}}_0$ to simplify notation. We now freely use the instruction "...any root of..." over $\mathbf{K}$, and we do not bother any more about computations in $\mathbf{K}$ except for the following:

— Some remarks following the description of Algorithms I and II, since the original descriptions of these algorithms did not use dynamic evaluation (actually, the idea that dynamic evaluation is a very general tool is quite new; cf. [DDD] as the first reference).

— The description of a method to compare different irreducible factors of $F(X, Y)$ at the end of § 1, as an example of dynamic evaluation "of level 2."

— The description of the normalization algorithm in § 5 where, at some points in the algorithm, we have to use explicitly the set of values and polynomials returned by dynamic evaluation.

The polynomial $F(X, Y)$. As explained above, the bivariate polynomial $F(X, Y)$ has its coefficients in some computable subfield $\mathbf{K}_0$ of $\mathbf{K}$. Since $F(X, Y)$ will often be considered as a polynomial in Y with coefficients in $\mathbf{K}[X]$, we denote

$$F(X, Y) = \sum_{i=0}^{n} a_i(X) Y^i$$

where n is the degree of F in Y.

We may assume that $F(X, Y)$ is square-free: Using the Gauss lemma [La, Chap. 5] and some classical algorithm like Yun's [Yu] for square-free decomposition in the Euclidean ring $\mathbf{K}_0(X)[Y]$, we easily get the following result.

PROPOSITION. *Using only derivations and gcd's computations in $\mathbf{K}_0[X]$ and in $\mathbf{K}_0(X)[Y]$, we may compute square-free polynomials $F_i(X, Y)$ in $\mathbf{K}_0[X, Y]$, pairwise coprime, and positive integers k_i such that*

$$F(X, Y) = \prod_{i=1}^{\kappa} F_i(X, Y)^{k_i}.$$

Of course the absolute factorization of F immediately follows from the absolute factorizations of the F_i's.

In addition, we may assume that we know some $\alpha \in \mathbf{K}_0$ such that $F(\alpha, T)$ is square-free and of degree n. Such an α is called a *noncritical value for* $F(X, Y)$. It means that the curve C has no vertical asymptote of equation $x = \alpha$ and that for every $\beta \in \mathbf{K}$ if (α, β) is a point of C it is *simple* (i.e., the derivatives $F'_X(X, Y)$ and $F'_Y(X, Y)$ of $F(X, Y)$ are not both zero at (α, β)) and without vertical tangent. In order to determine α, we may simply test successively $\alpha = 0, 1, -1, 2, -2, \cdots$ until one is noncritical. It must happen: Let $D(X) \in \mathbf{K}_0[X]$ be the discriminant of $F(X, Y)$ (considered as a polynomial in Y). It is well known that α is noncritical for $F(X, Y)$ if and only if $a_n(\alpha)D(\alpha) \neq 0$. But $a_n(X)D(X)$ has a finite number of roots (because

$F(X, Y)$ is square-free), and the image of the integers in $\mathbf{K}_0$ is infinite (because $\mathbf{K}_0$ has characteristic zero), whence a noncritical value of α must be found after a finite number of trials (and usually in practice a very short number of them). In addition, in Kaltofen's algorithm we assume that $\alpha=0$, in order to simplify the description of the algorithm. For this purpose, if zero is a critical value for $F(X, Y)$ and $\alpha \in \mathbf{K}_0$ a noncritical one, we replace $F(X, Y)$ by $F(X+\alpha, Y)$.

Now, if $P(T)=F(\alpha, T) \in \mathbf{K}_0[T]$, and if $\beta \in \mathbf{K}$ is any root of $P(T)$, the point (α, β) is a simple point on the curve C. As a consequence, there is one and only one irreducible component of C through this point, i.e., one and only one absolutely irreducible factor $\Phi(X, Y)$ of $F(X, Y)$ such that $\Phi(\alpha, \beta)=0$. The three algorithms below return this factor $\Phi(X, Y)$. The subfield $\mathbf{K}_1=\mathbf{K}_0(\beta)$ of $\mathbf{K}$ generated by $\mathbf{K}_0$ and β is a finite extension of $\mathbf{K}_0$ of degree at most n.

In order to make the description of the algorithms easier, we may also require that $F(X, Y)$ be either *primitive in* Y (i.e., $\gcd(a_i(X))_i=1$) or even *monic in* Y (i.e., $a_n(X)=1$). In the first case we replace $F(X, Y)$ by $F(X, Y)/\gcd(a_i(X))_i$. In the second case, using a classical trick, we may replace $F(X, Y)$ by $G(X, Y)=\sum_i b_i(X) Y^i \in \mathbf{K}_0[X, Y]$, where $b_n(X)=1$ and $b_i(X)=a_i(X) a_n(X)^{n-1-i}$ for $1 \leqq i \leqq n-1$. Then $G(X, Y)$ is monic in Y and is related to $F(X, Y)$ by $G(X, a_n(X) Y)= a_n(X)^{n-1} F(X, Y)$, so that the absolute factorization of $F(X, Y)$ is easily obtained from that of $G(X, Y)$, and $G(0, T)$ is square-free of degree n if $F(0, T)$ is. Monicity may also be enforced by making a linear change of coordinates $X'=X+aY$ for a "random" $a \in \mathbf{K}$, but the degree in Y of the resulting polynomial is the total degree d of $F(X, Y)$, which may be much higher than n.

With either of these two assumptions we may compute in the Euclidean ring $\mathbf{K}(X)[Y]$ rather than in $\mathbf{K}[X, Y]$, which is not Euclidean, by the Gauss lemma [La, Chap. 5]. Especially, if $F(X, Y)$ is primitive (respectively, monic) in Y and if $\Psi(X, Y)$ is an irreducible factor of $F(X, Y)$ in $\mathbf{K}(X)[Y]$, then multiplying $\Psi(X, Y)$ by the suitable fraction in $\mathbf{K}(X)$ to make it primitive (respectively, monic) in Y gives an irreducible factor $\Phi(X, Y)$ of $F(X, Y)$ in $\mathbf{K}[X, Y]$.

We now come to the description of the three algorithms, each one followed by its application to the factorization (over $\mathbf{K}=\mathbf{C}$) of

$$F_q(X, Y)=Y^2-q(X+1)^2=(Y-\sqrt{q}\, X-\sqrt{q})(Y+\sqrt{q}\, X+\sqrt{q})$$

for any rational number $q \neq 0$. It is a square-free polynomial with coefficients in the computable field $\mathbf{K}_0=\mathbf{Q}$, monic in Y, and $F_q(0, T)=T^2-q$ is square-free, so that we may choose $\alpha=0$ and $P(T)=T^2-q$.

ALGORITHM I (described by Trager [Trag2], and in the last section of [Trav]—but independently from the preceding sections).

INITIALIZE. Assume $F(X, Y)$ is square-free in $\mathbf{K}_0[X, Y]$ (as above).

SIMPLE POINT. Let $\alpha \in \mathbf{K}_0$ be a noncritical value for $F(X, Y)$ (as above). Let $P(T)=F(\alpha, T)$, let β be any root of $P(T)$ in $\mathbf{K}$, and let $\mathbf{K}_1$ be the subfield $\mathbf{K}_0(\beta)$ of $\mathbf{K}$.

RATIONAL FACTORIZATION. Factorize $F(X, Y)$ in $\mathbf{K}_1[X, Y]$.

RESULT. Now $\Phi(X, Y)$ is the unique factor of $F(X, Y)$ in $\mathbf{K}_1[X, Y]$ such that $\Phi(\alpha, \beta)=0$.

This algorithm is very easy to describe. With this point of view, absolute factorization is a special case of rational factorization. It means that we do not only need that $\mathbf{K}_0$ be a computable field. In addition, we require an algorithm to factorize bivariate polynomials over simple algebraic extensions of $\mathbf{K}_0$, like $\mathbf{K}_1$. However, when $\mathbf{K}_0$ is $\mathbf{Q}$,

for example, algorithms are known for rational factorization in $\mathbf{K}_1[X, Y]$. But they are quite complicated and slow, especially if the degree of $\mathbf{K}_1$ over $\mathbf{K}_0$ is high.

In the original description, two more rational factorizations were required. At initialization, $F(X, Y)$ was assumed irreducible in $\mathbf{K}_0[X, Y]$, whence a first factorization. However, applying Traverso's criterion in [Trav] to $\Phi(X, Y) \in \mathbf{K}_1[X, Y]$ proves that this assumption is useless. The factorization of $P(T)$ in $\mathbf{K}_0[T]$ was also required, in order to be able to compute in $\mathbf{K}_1$. Strictly speaking, it should still be required, since it has not yet been proven that it is possible to factorize polynomials over $\mathbf{K}_1$ when the minimal polynomial of β over $\mathbf{K}_0$ is not known. Indeed, the usual algorithms for rational factorization over $\mathbf{K}_1$ do not only use equality tests and basic field operations in $\mathbf{K}_1$, they also use operations on some finite field, including the Frobenius map on that field. However, we conjecture that it is possible to run the classical rational factorization algorithms over $\mathbf{K}_1$ without factoring $P(T)$ in $\mathbf{K}_0[T]$. Note that if we do not factorize $F(X, Y)$ over $\mathbf{K}_0$ we cannot use the following nice result (cf. [CG], for example): If $F(X, Y)$ is irreducible over $\mathbf{K}_0$ and if by chance $P(T)$ has a linear factor in $\mathbf{K}_0[T]$ then $F(X, Y)$ is absolutely irreducible.

Example. The factorization of $F_q(X, Y)$ over $\mathbf{K}_1$ is $Y^2 - q(X+1)^2 = (Y - \beta X - \beta)(Y + \beta X + \beta)$, and the result is $\Phi(X, Y) = Y - \beta X - \beta$. Here the absolute factorization of $F_q(X, Y)$ is the same as its rational factorization over $\mathbf{K}_1$, but in general it is false (consider, for example, $Y^3 - 2X^3$).

ALGORITHM II (Kaltofen [Ka]). This algorithm replaces the rational factorization in $\mathbf{K}_1[X, Y]$ in Algorithm I by an ad hoc method for the determination of the unique irreducible factor $\Phi(X, Y)$ through (α, β). It has been developed independently from Algorithm I, and its presentation used to be fairly different.

INITIALIZE. Assume $F(X, Y)$ is monic in Y and square-free (as above).

SIMPLE POINT. Assume that zero is a noncritical value for $F(X, Y)$ (as above). Let $P(T) = F(0, T)$, let β be any root of $P(T)$ in $\mathbf{K}$, and let $\mathbf{K}_1$ be the subfield $\mathbf{K}_0(\beta)$ of $\mathbf{K}$.

A ROOT OF F. By the (formal) implicit function theorem, there exists a unique $y = \beta + \sum_{i=1}^{\infty} \beta_i X^i$ in $\mathbf{K}_1[[X]]$ such that $F(X, y) = 0$ in $\mathbf{K}_1[[X]]$, i.e., such that y is a root of $F(X, Y)$, considered as a polynomial in Y, in $\mathbf{K}_1[[X]]$. The computation of the β_i's successively for $i = 1, 2, \cdots$ is easy, it is linear algebra over $\mathbf{K}_1$. In this step, compute $\beta_1, \beta_2, \cdots, \beta_k$ for $k = (2n-1)m$ (where $m = \deg_X(F)$).

RESULT. Now $\Phi(X, Y)$ is the minimal polynomial of y over $\mathbf{K}_1[X]$. It means that $\Phi(X, y) = 0$ and that $\Phi(X, Y) = \varphi_0(X) + \varphi_1(X)Y + \cdots + \varphi_{l-1}(X)Y^{l-1} + Y^l$ with the $\varphi_i(X)$'s in $\mathbf{K}_1[X]$ of degree at most m and with l as small as possible $(1 \leqq l \leqq n)$. If l is given, the determination of the coefficients of the $\varphi(X)$'s is just linear algebra over $\mathbf{K}_1$ and only uses β_i for $i = 1, 2, \cdots, k$. Since l is not known, try $l = 1, 2, \cdots, n-1$ until you find a solution. If none is found then $\Phi(X, Y) = F(X, Y)$.

Example. The implicit function theorem gives the root $y = \beta + \beta X$, with minimal polynomial $\Phi(X, Y) = Y - \beta X - \beta$ of degree $l = 1$.

In his paper Kaltofen does not use general results from dynamic evaluation to prove that we do not need to factorize $P(T)$ in $\mathbf{K}_0[T]$. However his argumentation is similar, but is restricted to the linear algebra that he needs. By doing so, he is able to get polynomial bounds on both the number of rational operations and on the size of the involved numerators and denominators, which we do not do for Algorithm III. In addition, Kaltofen remarks that no splitting should occur in the computation of the root y of F, since this computation does not require any equality test, and only divisions

by $P'(0)$ which is invertible in the ring $\mathbf{K}_0[T]/(P(T))$. This remark will be useful to us in a fairly different context in § 5.

ALGORITHM III. Algorithm III is proved in the rest of this paper. Notation and terminology will be defined and described in other sections. As usual, C denotes the curve in $\mathbf{A}_2(\mathbf{K})$ of equation $F(x, y) = 0$. Let us just say here that:

— $\mathbf{K}_0(C)$ is the ring $\mathbf{K}_0(X)[Y]/(F(X, Y))$ and similarly $\mathbf{K}_1(C)$ is the ring $\mathbf{K}_1(X)$-$[Y]/(F(X, Y))$ (the images of X and Y in these rings are respectively denoted x and y),

— $L(0)$ is a $\mathbf{K}$-vector space (associated to C) of finite dimension precisely equal to the number r of absolutely irreducible factors of $F(X, Y)$,

— each simple point on C corresponds to a so-called place of $\mathbf{K}(C)$, and to each place $\wp$ of $\mathbf{K}(C)$ is associated a subspace $L(-\wp)$ of $L(0)$ of dimension $r-1$.

We need subalgorithms for the computation of a basis of the spaces $L(0)$ and $L(-\wp)$. Such algorithms are described in § 5.

INITIALIZE. Assume $F(X, Y)$ is primitive in Y and square-free (as above).

SPACE $L(0)$. Compute a basis B in $\mathbf{K}_0(C)$ of $L(0)$. Let r denote the cardinal of B, i.e., the dimension of $L(0)$ over $\mathbf{K}$. It is the number of absolutely irreducible factors of $F(X, Y)$. If $r = 1$ then $F(X, Y)$ is absolutely irreducible, return $\Phi(X, Y) = F(X, Y)$ and exit.

SIMPLE POINT. Let $\alpha \in \mathbf{K}_0$ be a noncritical value for $F(X, Y)$ (as above). Let $P(T) = F(\alpha, T)$, let β be any root of $P(T)$ in $\mathbf{K}$, and let $\mathbf{K}_1$ be the subfield $\mathbf{K}_0(\beta)$ of $\mathbf{K}$.

SPACE $L(-\wp)$. Let $\wp$ be the place which corresponds to the simple point (α, β). Compute from B a basis B' in $\mathbf{K}_1(C)$ of $L(-\wp)$.

RESULT. If we normalize the gcd of polynomials in $\mathbf{K}_1(X)[Y]$ so that it is primitive in Y, then $\Phi(X, Y)$ is the gcd in $\mathbf{K}_1(X)[Y]$ of $F(X, Y)$ and of representatives of the elements of B'.

It will be clear from the rest of the paper that most work here is made for the computation of the basis of $L(0)$. Another difference between this algorithm and the other two is that the number r of factors is computed before any explicit factor, so that the computation stops if $r = 1$.

Example. In the example, $B = \{1, y/(x+1)\}$ so that $r = 2$. As usual let $\alpha = 0$; then $B' = \{-\beta + y/(x+1)\}$ and $\Phi(X, Y) = \gcd(F_q(X, Y), Y - \beta(X+1)) = Y - \beta X - \beta$.

Other factors. There is still one point to discuss, which is common to the three algorithms described here: once one factor $\Phi(X, Y)$ is obtained, how do we get the complete factorization? For simplicity, we assume here that $\Phi(X, Y)$ is normalized in some way, for example, assume that it is monic in Y. These algorithms return a factor $\Phi_\beta(X, Y)$ for each root β of $P(T)$, and each absolutely irreducible factor of $F(X, Y)$ is one of the $\Phi_\beta(X, Y)$'s. Two values of β may correspond to the same factor, precisely when the two points (α, β) lie on the same irreducible component of the curve C. It is possible to get rid of redundant factors by testing whether $\Phi_{\beta_1}(X, Y)$ and $\Phi_{\beta_2}(X, Y)$ are equal in $\mathbf{K}_0(\beta_1, \beta_2)[X, Y]$ for $\beta_1 \neq \beta_2$. Here we have to test equalities in the field $\mathbf{K}_0(\beta_1, \beta_2)$ where β_1 and β_2 represent any root of two factors $P_1(T)$ and $P_2(T)$ of $P(T)$ in $\mathbf{L}[T]$ which are either coprime or equal. For this purpose, we first use the instruction:

"Let β_1 be any root of $P_1(T)$ in $\mathbf{K}[T]$"

and then in the "coprime" case the instruction:

"Let β_2 be any root of $P_2(T)$ in $\mathbf{K}[T]$"

and in the "equal" case, to ensure that $\beta_2 \neq \beta_1$, the instruction:

"Let β_2 be any root of $P_2(T)/(T-\beta_1)$ in $\mathbf{K}[T]$"

Example. For $F_q(X, Y)$ we have $\Phi_\beta(X, Y) = Y - \beta X - \beta$ for every root β of $P(T) = T^2 - q$. Let β' denote the other root of $P(T)$, i.e., the root of $P(T)/(T-\beta) = T+\beta$. Then $\Phi_{\beta'}(X, Y) = Y - \beta' X - \beta' = Y + \beta X + \beta$. Both polynomials have the same degree and are monic in Y but they are not equal (because if $\beta = -\beta$, then T would divide $P(T)$). So they are coprime and the complete absolute factorization of $F_q(X, Y)$ is $(Y - \beta X - \beta)(Y + \beta X + \beta)$.

Rational factorization. A related question is whether it is possible to derive the factorization of $F(X, Y)$ in $\mathbf{K}_0[X, Y]$ from its absolute factorization. It does not seem very interesting in practice, although the answer is yes, but here rational factorization of $P(T)$ is needed. Let $\Phi(X, Y) \in \mathbf{K}_1[X, Y]$ be the absolutely irreducible factor of $F(X, Y)$ such that $\Phi(\alpha, \beta) = 0$ as above, and let $Q(T)$ be the minimal polynomial of β over $\mathbf{K}_0$. Define $\Psi(T, X, Y) \in \mathbf{K}_0[T, X, Y]$ (with degree in T less than $\deg(Q(T))$) such that $\Phi(X, Y) = \Psi(\beta, X, Y)$. Then the *resultant* $R(X, Y)$ of $\Psi(T, X, Y)$ and $Q(T)$ (considered as polynomials in T) is in $\mathbf{K}_0[X, Y]$, and it is a power of an irreducible factor of $F(X, Y)$ in $\mathbf{K}_0[X, Y]$ [Trag1]. This factor can be obtained as the gcd of $F[X, Y]$ and $R(X, Y)$ in $\mathbf{K}_0(X)[Y]$.

About the characteristic. The assumption that $\mathbf{K}$ has characteristic zero can easily be weakened. It is clear that the three algorithms run the same way if $\mathbf{K}_0$ is a "large enough" finite field. It means that it must be possible to find some noncritical value α for $F(X, Y)$ in $\mathbf{K}_0$. If $\mathbf{K}_0$ is too small then a first extension may be used to find some noncritical α. In addition, for Algorithm III, the characteristic should be larger than the degree n of F in Y in order to use Puiseux expansions as in this paper. This remark is important by itself, since, for example, algebraic curves over finite fields are studied in coding theory [Go], and also in light of the remark in the conclusion about the use of arithmetic modulo p in order to control the size of the coefficients in the algorithm when used over the field of rational numbers.

2. Some geometry. In this section, as is often the case in algebraic geometry, but not as in the other sections of this paper, we assume that the polynomial $F(X, Y)$ is irreducible in $\mathbf{K}[X, Y]$. Its degree n in Y is assumed positive. Our references are Fulton's book [Fu] and Walker's [Wa].

Functions. The $\mathbf{K}$-algebra $\mathbf{K}(C) = \mathbf{K}(X)[Y]/(F(X, Y))$ is a *field* because $F(X, Y)$ is irreducible in $\mathbf{K}[X, Y]$. This field is a finite algebraic extension of $\mathbf{K}(X)$ of degree n. Every element f of $\mathbf{K}(C)$ is the root of one and only one monic irreducible polynomial with coefficients in $\mathbf{K}(X)$, called its *minimal polynomial over* $\mathbf{K}(X)$. If the coefficients of this polynomial are in $\mathbf{K}[X]$, then f is *integral* (over $\mathbf{K}[X]$). The set A of integral elements of $\mathbf{K}(C)$ is a free $\mathbf{K}[X]$-algebra of rank n. Its field of fractions is $\mathbf{K}(C)$, which is also equal to the tensor product $\mathbf{K}(X) \otimes_{\mathbf{K}[X]} A$. This means that every element in $\mathbf{K}(C)$ can be written f/g for some f in a and some $g \neq 0$ in $\mathbf{K}[X]$.

From a geometric point of view, if C is the affine curve of equation $F(x, y) = 0$, then $\mathbf{K}(C)$ is called the *function field* of C. It comes from the fact that it is the field of fractions of the integral domain $\mathbf{K}[X, Y]/(F(X, Y))$, which in turn is made of the polynomials of $\mathbf{K}[X, Y]$ modulo the equivalence relation "have the same value at each point of C." For example, $\mathbf{K}(X)$ is the function field of the curve C_0 of equation $y = 0$, i.e., the "x-axis," and its ring of integral elements if $\mathbf{K}[X]$. The curve C_0 can be identified with the affine line $\mathbf{A}_1(\mathbf{K})$.

The *projective completion* $\hat{C}$ of C in the projective plane $\mathbf{P}_2(\mathbf{K})$ is now defined as the set of points $(x, y, z) \in \mathbf{P}_2(\mathbf{K})$ such that $\hat{F}(x, y, z) = 0$, where $\hat{F}(X, Y, Z) = Z^d F(X/Z, Y/Z)$ is homogenous of degree d, and d is the total degree of $F(X, Y)$.

The curve $\hat{C}$ has a finite number of points at infinity, often called the *points at infinity of C*, which correspond to the *asymptotic directions* of C. For example, the projective completion of C_0 is identified with the projective line $\mathbf{P}_1(\mathbf{K})$. The point $(1, 0, 0)$ is the only point at infinity of C_0, and will be denoted ∞.

Places. Following Walker [Wa, Chap. 4], we define the *places* of the curve $\hat{C}$, or of the field $\mathbf{K}(C)$, as the equivalence classes of irreducible parametrizations of $\hat{C}$. We refer to Walker for precise definitions of irreducibility and equivalence. A parametrization of $\hat{C}$ is a pair $(\varphi(t), \psi(t))$ of power series in $\mathbf{K}((t))$ such that the power series $F(\varphi(t), \psi(t))$ is equal to zero in $\mathbf{K}((t))$. To each point $P = (x, y, z)$ of $\hat{C}$ are associated a finite number of places *centered* at that point, which correspond bijectively to the "branches" of $\hat{C}$ through P (a place is made of the irreducible parametrizations of the corresponding branch). If $P = (x, y, 1)$ is a point of C the places centered at P are represented by a parametrization $(\varphi(t), \psi(t))$ with both series in $\mathbf{K}[[t]]$ and $\varphi(0) = x$, $\psi(0) = y$.

Since there is exactly one branch of $\hat{C}$ through every simple point of C, such a point is the center of exactly one place. The other points of $\hat{C}$ are called *singular*, they are in finite number, and each singular point may be the center of several places. For example, since every point of the projective line $\hat{C}_0$ is simple, we identify each place of $\hat{C}_0$, or of $\mathbf{K}(X)$, with its center. So that the places of $\hat{C}_0$ are the points of $\hat{C}_0 = \mathbf{P}_1(\mathbf{K}) = \mathbf{K} \cup \{\infty\}$.

Each place $\wp$ of $\hat{C}$ is *above* one place α of $\hat{C}_0$, and we note $\wp \mid \alpha$. Precisely, if the center $P = (x, y, z)$ of $\wp$ is a point $(x, y, 1)$ of C, then $\alpha = x$. If P is a point $(x, y, 0)$ (i.e., the corresponding branch is an infinite branch of C), then $\alpha = \infty$ except when this branch is a vertical asymptote of equation $x = x_0$, in which case $\alpha = x_0$.

In order to compute with places, we shall choose some precise parametrization to represent each place, using rational Puiseux expansions of $\hat{C}$ (cf. § 5).

Let $\wp$ be a place of $\mathbf{K}(C)$, represented by the parametrization $(\varphi(t), \psi(t))$. To every polynomial $G(X, Y)$ in $\mathbf{K}[X, Y]$ is associated a power series $G(\varphi(t), \psi(t))$ in $\mathbf{K}((t))$. In fact this series only depends on the image of $G(X, Y)$ in $\mathbf{K}(C)$, and thus by extension of scalars to $\mathbf{K}(X)$ we get a $\mathbf{K}$-homomorphism $f \mapsto f \circ (\varphi, \psi)$ from $\mathbf{K}(C)$ in $\mathbf{K}((t))$, which is injective. Now, let f be a nonzero function of $\mathbf{K}(C)$, and let $f \circ (\varphi, \psi) = \sum_{k=\nu}^{+\infty} u_k t^k$ with $u_\nu \neq 0$. The integer ν does not depend on the choice of the parametrization for $\wp$. We say that f has *order* ν at the place $\wp$, and we denote $w_\wp(f) = \nu$. Also let $w_\wp(0) = +\infty$.

If $\nu > 0$, we say that f has a *zero of order* ν at $\wp$, and if $\nu < 0$, that f has a *pole of order* $-\nu$ at $\wp$. It can be proved that the integral elements of $\mathbf{K}(C)$ are the functions on C which have no pole at the places of $\mathbf{K}(C)$ that are not above ∞. It can also be proved that a function on C has a finite number of zeros and poles on $\hat{C}$, and that

$$\sum_{\wp} w_\wp(f) = 0$$

where the sum is over all the places of $\mathbf{K}(C)$.

If C is the x-axis C_0, the order of f at α is denoted $v_\alpha(f)$, for every f in $\mathbf{K}(C_0)$ and every α in $\mathbf{K} \cup \{\infty\}$. If $\alpha \in \mathbf{K}$ then $v_\alpha(X - \alpha) = 1$ and for every place $\wp$ of $\hat{C}$ above α the positive integer $e_\wp$ such that $w_\wp(X - \alpha) = e_\wp$ is called the *ramification index* of the place $\wp$. Similarly, if $\alpha = \infty$ then $v_\infty(1/X) = 1$ and for every place $\wp$ of $\hat{C}$ above ∞ the positive integer $e_\wp$ such that $w_\wp(1/X) = e_\wp$ is called the *ramification index* of the place $\wp$. For every $\alpha \in \mathbf{K} \cup \{\infty\}$ we have the relation $\sum_{\wp \mid \alpha} e_\wp = n$.

Divisors. The group $\mathscr{D}(\hat{C})$ of the *divisors* on the curve $\hat{C}$ may now be defined. It is made of the formal expressions $D = \sum_\wp n_\wp \wp$ where the sum is over all the places $\wp$

of $\hat{C}$, the $n_\wp$ are integers, and all but a finite number of them are equal to zero. Equivalently, $\mathscr{D}(\hat{C})$ is the free Abelian group on the set of places of $\mathbf{K}(C)$. To each function f of $\mathbf{K}(C)$, different from zero, is associated the divisor $\operatorname{div}(f) = \operatorname{div}_C(f) = \sum_\wp w_\wp(f)\wp$. A partial order is defined on $\mathscr{D}(\hat{C})$ by

$$\sum_\wp n_\wp \wp \geqq \sum_\wp n'_\wp \wp \Leftrightarrow \forall \wp,\ n_\wp \geqq n'_\wp.$$

In addition, let $\operatorname{div}(0) = +\infty$, and $\bar{\mathscr{D}}(\hat{C}) = \mathscr{D}(\hat{C}) \cup \{+\infty\}$, with the order obtained by adding to $\geqq$ on $\mathscr{D}(\hat{C})$ the rule

$$+\infty \geqq D \quad \text{for every } D.$$

The elements of $\bar{\mathscr{D}}(\hat{C})$ are called here the *generalized divisors* on $\hat{C}$.

Notation. For every divisor D on $\hat{C}$, the set of functions f in $\mathbf{K}(C)$ such that $\operatorname{div}(f) \geq -D$ is denoted $L(D)$ or sometimes $L_C(D)$.

This set is a $\mathbf{K}$-vector space. For example, $L(0)$ is the set of functions in $\mathbf{K}(C)$ which are integral and which have no pole at the places of $\mathbf{K}(C)$ above ∞. Our factorization method is based on the two following classical results (cf. [Fu, Chap. 8]).

THEOREM 1. *For every divisor D on $\hat{C}$, $L(D)$ is a finite-dimensional $\mathbf{K}$-vector space.*

THEOREM 2. $L(0) = \mathbf{K}$.

3. Number of factors. In this section, the polynomial $F(X, Y)$ in $\mathbf{K}[X, Y]$ is no more assumed irreducible, but simply square-free and primitive in Y. Let $F(X, Y) = \prod_{i=1}^r F_i(X, Y)$ be the irreducible decomposition of $F(X, Y)$ in $\mathbf{K}[X, Y]$. We now generalize the definitions and results of § 2.

Functions. A first point is that the $\mathbf{K}(X)$-algebra of functions on C

$$\mathbf{K}(C) = \mathbf{K}(X)[Y]/(F(X, Y))$$

still has degree n over $\mathbf{K}(X)$, but is no more a field. However, by the Chinese Remainder Theorem, it is isomorphic to the product of fields $\prod_{i=1}^r \mathbf{K}(C_i)$, where $\mathbf{K}(C_i) = \mathbf{K}(X)[Y]/(F_i(X, Y))$. Precisely, let Θ_i denote the projection of $\mathbf{K}(C)$ on $\mathbf{K}(C_i)$ for each i, and

$$\Theta = \prod_{i=1}^r \Theta_i : \mathbf{K}(C) \to \prod_{i=1}^r \mathbf{K}(C_i)$$

the isomorphism in the Chinese Remainder Theorem. Let A_i be the ring of integral elements of $\mathbf{K}(C_i)$ for $1 \leqq i \leqq r$, and let us define the ring of integral elements A of $\mathbf{K}(C)$ by

$$A = \Theta^{-1}\left(\prod_{i=1}^r A_i\right).$$

It means that A is made of the functions f of $\mathbf{K}(C)$ which are integral on each component of C. Since A is isomorphic to the product of the A_i's, it is a free $\mathbf{K}[X]$-module of rank n, and $\mathbf{K}(C) = \mathbf{K}(X) \otimes_{\mathbf{K}[X]} A$.

The projective completion $\hat{C}$ of C is defined as the set of points (x, y, z) in $\mathbf{P}_2(\mathbf{K})$ such that $\hat{F}(x, y, z) = 0$ exactly as in § 2. It is easy to prove that $\hat{C}$ is the union of the projective completions $\hat{C}_i$'s of the C_i's.

Places. It is still possible to define the places of $\hat{C}$ as its equivalence classes of irreducible parametrizations, and to represent them by rational Puiseux expansions, since $F(X, Y)$ is square-free and primitive in Y, i.e., C contains no multiple component and no vertical line. The set of places of $\hat{C}$ is then equal to the *sum* of the sets of

places of the curves C_i's, which means that each place $\wp$ of $\hat{C}$ is a place of one and only one irreducible component $\hat{C}_i$, called the *support* of $\wp$.

If $\wp$ is a place of $\hat{C}$ represented by some parametrization $(\varphi(t), \psi(t))$, the $\mathbf{K}$-homomorphism $f \mapsto f \circ (\varphi, \psi)$ from $\mathbf{K}(C)$ to $\mathbf{K}((t))$ is defined as if C were irreducible.

LEMMA. *Let f be a function of $\mathbf{K}(C)$ and $\wp$ a place of $\hat{C}$ represented by $(\varphi(t), \psi(t))$. Let $\hat{C}_i$ be the support of $\wp$. Then*

$$f \circ (\varphi, \psi) = \Theta_i(f) \circ (\varphi, \psi).$$

Proof. If f is the image of a polynomial $G(X, Y)$ of $\mathbf{K}[X, Y]$ modulo $F(X, Y)$, then $\Theta_i(f)$ is the image of G modulo F_i, and thus both $f \circ (\varphi, \psi)$ and $\Theta_i(f) \circ (\varphi, \psi)$ are equal to $G(\varphi(t), \psi(t))$. The lemma follows by extension of scalars to $\mathbf{K}(X)$.

It is thus possible to define the order of a function f of $\mathbf{K}(C)$ at any place $\wp$ of $\hat{C}$ as the t-order of the series $f \circ (\varphi, \psi)$, and it is equal to the order of the function $\Theta_i(f)$ of $\mathbf{K}(C_i)$ at $\wp$ considered as a place of $\hat{C}_i$. It follows that a function of $\mathbf{K}(C)$ is integral if and only if its order is nonnegative at each place of $\hat{C}$ which does not lie above ∞.

Divisors. We define the group of *divisors* on $\hat{C}$ as the direct product

$$\mathscr{D}(\hat{C}) = \prod_{i=1}^{r} \mathscr{D}(\hat{C}_i).$$

Thus, it is the free Abelian group on the places of $\hat{C}$. Its elements are still denoted $D = \sum_{\wp} n_{\wp}\wp$. The set of *generalized divisors* on $\hat{C}$ is defined as the Cartesian product

$$\bar{\mathscr{D}}(\hat{C}) = \prod_{i=1}^{r} \bar{\mathscr{D}}(\hat{C}_i).$$

The order $\geqq$ is defined on this set from the orders $\geqq$ on the sets $\bar{\mathscr{D}}(\hat{C}_i)$'s by

$$(D_i)_{1 \leqq i \leqq r} \geqq (D'_i)_{1 \leqq i \leqq r} \Leftrightarrow \forall i,\ D_i \geqq D'_i.$$

Of course, $\mathscr{D}(\hat{C})$ is a subset of $\bar{\mathscr{D}}(\hat{C})$.

For every function f on $\mathbf{K}(C)$, we denote by $\operatorname{div}(f)$ or $\operatorname{div}_C(f)$ the generalized divisor on C

$$\operatorname{div}_C(f) = (\operatorname{div}_{C_i}(\Theta_i(f)))_{1 \leqq i \leqq r}.$$

It is a divisor on $\hat{C}$ if and only if $\Theta_i(f)$ is a nonzero function in $\mathbf{K}(C_i)$ for each i.

The space $L(D)$, or $L_C(D)$, may now be defined for any divisor D of $\hat{C}$ as the set of functions f in $\mathbf{K}(C)$ such that $\operatorname{div}(f) \geqq -D$.

THEOREM 3. *Let $D = (D_i)_{1 \leqq i \leqq r}$ be a divisor on $\hat{C}$. Then Θ defines an isomorphism between $L_C(D)$ and the product $\prod_{r=1}^{r} L_{C_i}(D_i)$.*

Proof. Let f denote a function of $\mathbf{K}(C)$, and $\wp$ a place of $\hat{C}$. Let $\hat{C}_i$ be the support of $\wp$. Since $w_{\wp}(f) = w_{\wp}(\Theta_i(f))$, the function f is in $L_C(D)$ if and only if for each i we have $\operatorname{div}_{C_i}(\Theta_i(f)) \geqq -D_i$, i.e., if and only if for each i the function $\Theta_i(f)$ is in $L_{C_i}(D_i)$.

Number of factors. Our main result now follows directly from Theorem 2.

RESULT 1. *The number of irreducible factors of $F(X, Y)$ in $\mathbf{K}[X, Y]$ is equal to the dimension of the $\mathbf{K}$-vector space $L_C(0)$.*

Remark. For $i = 1$ to r, let h_i denote the function of $\mathbf{K}(C)$ such that $\Theta(h_i) = (\delta_{j,i})_{1 \leqq j \leqq r}$, where $\delta_{j,i}$ is one if $j = i$ and zero otherwise. Then $\{h_i\}_{1 \leqq i \leqq r}$ is a basis of $L_C(0)$ over $\mathbf{K}$. The functions h_i's are related to the irreducible factors F_i's of F in the classical way. Let $F_i^*(X, Y) = \prod_{j \neq i} F_j(X, Y) = F(X, Y)/F_i(X, Y)$, and let $G_i(X, Y)$

denote a polynomial with image modulo $F_i(X, Y)$ equal to the inverse of the image of $F_i^*(X, Y)$. If $H_i(X, Y)$ denotes the product $F_i^*(X, Y)G_i(X, Y)$, then h_i is the image of $H_i(X, Y)$ modulo $F(X, Y)$.

4. Determination of a factor. Once the number of irreducible factors of $F(X, Y)$ in $\mathbf{K}[X, Y]$ is known, how is it possible to compute one of them? A trivial remark is that there is nothing to do when the number of irreducible factors is one. This remark is important, since in other factorization methods it may take very long to recognize irreducible polynomials.

A method for computing the irreducible factors is now described, which uses a second consequence of Theorem 3.

RESULT 2. *Let $\wp$ be a place of $\hat{C}$, and $\hat{C}_i$ the support of $\wp$. Then $L(-\wp)$ is the sub-$\mathbf{K}$-vector space of $L_C(0)$ of dimension $(r-1)$ and of basis the h_j's for $j \neq i$.*

Proof. Let $D=-\wp$, so that $D=(D_j)_{1\leqq j\leqq r}$ with $D_i=-\wp$ and $D_j=0$ for $j\neq i$. The space $L_{C_i}(-\wp)$ is made of the functions which are in $L_{C_i}(0)$ and in addition have a zero at $\wp$. Since $L_{C_i}(0)$ is made of the constant functions, it follows that $L_{C_i}(-\wp)$ is the null space $\{0\}$. Whence the result, by Theorem 3.

The following result describes how it is possible to compute an irreducible factor of $F(X, Y)$ from a basis of $L_C(-\wp)$.

PROPOSITION. *Let $\wp$ be a place of $\hat{C}$, and $\hat{C}_i$ the support of $\wp$. Let $\{b_1, b_2, \cdots, b_{r-1}\}$ be a $\mathbf{K}$-basis of $L_C(-\wp)$, and let $B_1(X, Y)$, $B_2(X, Y), \cdots, B_{r-1}(X, Y)$ be polynomials in $\mathbf{K}(X)[Y]$ such that $B_k(X, Y) \bmod F(X, Y)$ is equal to b_k for $1\leqq k\leqq r-1$. Then the irreducible factor $F_i(X, Y)$ of $F(X, Y)$ in $\mathbf{K}[X, Y]$ is the* gcd *of the polynomials $B_1(X, Y)$, $B_2(X, Y), \cdots, B_{r-1}(X, Y)$ and $F(X, Y)$ in $\mathbf{K}(X)[Y]$.*

Proof. Here the gcd of polynomials in $\mathbf{K}(X)[Y]$ is chosen primitive in Y. First, note that the gcd of $B_1, B_2, \cdots, B_{r-1}, F$ in $\mathbf{K}(X)[Y]$ does not depend on the choice of the B_k's; it only depends on the B_k's modulo F, i.e., on the b_k's.

Let us first assume that $\{b_1, b_2, \cdots, b_{r-1}\}$ is the basis of $L_C(-\wp)$ made of the h_j's for $j\neq i$. Then the $B_k(X, Y)$'s can be chosen as the $H_j(X, Y)$'s of § 3. But $H_j=F_j^*G_j$ where F_j^* divides F, $F/F_j^*=F_j$, and G_j is coprime to F_j. Thus, the gcd of F and H_j is F_j^*, and the gcd of F and of all the H_j's for $j\neq i$ is the gcd of all the F_j^*'s for $j\neq i$, i.e., it is F_i, as wanted.

Now, if $\{b_1, b_2, \cdots, b_{r-1}\}$ is any basis of $L_C(-\wp)$, let us denote

$$b_k=\sum_{j\neq i} m_{k,j}h_j$$

with the $m_{k,j}$'s in $\mathbf{K}$. Then we can choose

$$B_k(X, Y)=\sum_{j\neq i} m_{k,j}H_j(X, Y)$$

for $1\leqq k\leqq r-1$. Every common divisor of the $H_j(X, Y)$'s (for $j\neq i$) divides every $B_k(X, Y)$ (for $1\leqq k\leqq r-1$), and the converse is true since the matrix formed by the $m_{k,j}$'s is invertible. It follows that the gcd of $B_1, B_2, \cdots, B_{r-1}$ is equal to the gcd of the H_j's for $j\neq i$, and that the gcd of $B_1, B_2, \cdots, B_{r-1}, F$ is still equal to F_i.

At that point, terminology and notation in Algorithm III (§ 1) are defined, and it is proved that the algorithm returns an absolutely irreducible factor of $F(X, Y)$. But we have not yet described the two subalgorithms: computation of a basis B of $L(0)$ over $\mathbf{K}$, and then computation of a basis B' of $L(-\wp)$ over $\mathbf{K}$, with B in $\mathbf{K}_0(C)=\mathbf{K}_0(X)[Y]/(F(X, Y))$ and B' in $\mathbf{K}_1(C)=\mathbf{K}_1(X)[Y]/(F(X, Y))$. This will be done in the next section.

5. Computation of $L(D)$. The spaces $L(D)$ are basic objects of algebraic geometry of curves; they are used in the Riemann-Roch theorem [B1] as well as in integration

[Da], [Trag-2] and in coding theory [Go]. They have been studied for a long time, but usually for an irreducible curve. At the end of the 19th century at least two algorithms were known for the computation of a **K**-basis of any space $L(D)$ over an irreducible curve C:

- The "geometric" method of von Brill and Noether, using the notion of adjoint curves [BN], [Fu], [LR].
- The "arithmetic" method of Dedekind and Weber, which uses Puiseux expansions [DW], [Bl], [Co].
- In addition a new method for the computation of $L(D)$ when $D=0$ is given by Trager [Trag2], which can be viewed as a "global arithmetic" method (while the Dedekind-Weber method is "local").

For our factorization method, we need an algorithm that may be adapted to reducible curves (without multiple components and "vertical lines"), and that can be run with dynamic evaluation, which means that it should not require operations in fields other than the four basic operations and equality tests. It seems that this is the case for each of the three algorithms above, but the proof has only been written for the Dedekind-Weber algorithm [Du2].

In this section we first define rational Puiseux expansions of $\hat{C}$. We then describe the computation of a basis of $L(-\wp)$ from a basis of $L(0)$ (which is very easy), and finally we describe the Dedekind-Weber algorithm (improved by the use of rational Puiseux expansions) for the computation of $L(0)$. As in §§ 3 and 4, the polynomial $F(X, Y)$ is square-free and primitive in Y.

Rational Puiseux expansions. The rational Puiseux expansions of $\hat{C}$ give a "computable" description of the places of $\hat{C}$. Here we give their definition and some of their properties, referring to [Du3] for an algorithm to compute them. They lead to an "asymmetric" description of places, where x and y play different roles.

First, let us define a *triangular set of polynomials* over some subfield **L** of **K** as a set $TP=\{P_1(X_1), P_2(X_1, X_2), \cdots, P_I(X_1, X_2, \cdots, X_I)\}$ for some $I \geqq 0$ where each $P_i(X_1, X_2, \cdots, X_i)$ is a polynomial with coefficients in **L**, which is monic and of positive degree when considered as a polynomial in X_i. Let

$$TP^{(i)}=\{P_1(X_1), P_2(X_1, X_2), \cdots, P_i(X_1, X_2, \cdots, X_i)\}$$

for $i=0, 1, \cdots, I$, and let $\mathbf{L}\langle TP^{(i)}\rangle$ denote the ring defined recursively by $\mathbf{L}\langle TP^{(0)}\rangle=\mathbf{L}$ and $\mathbf{L}\langle TP^{(i)}\rangle=\mathbf{L}\langle TP^{(i-1)}\rangle[X_i]/(P_i(x_1, x_2, \cdots, x_{i-1}, X_i))$ if $i>0$, where x_j is the image of x_j in $\mathbf{L}\langle TP^{(j')}\rangle$ for $j=1, 2, \cdots, I$ and $1 \leqq j \leqq j'$. The *degree* of TP is $d(TP)=\prod_{1\leqq i\leqq I} d_i$ where d_i is the degree of P_i as a polynomial in x_i. It is the dimension of $\mathbf{L}\langle TP\rangle$ as a vector space over **L**. A basis $B(\mathbf{L}\langle TP\rangle)$ of $\mathbf{L}\langle TP\rangle$ over **L** is the set of monomials $\prod_{1\leqq i\leqq I} x_i^{l_i}$ with $l_i \in \{0, 1, \cdots, d_i-1\}$.

A *solution* of TP is a set $S=\{\beta_1, \beta_2, \cdots, \beta_I\}$ of elements of **K** such that $P_i(\beta_1, \beta_2, \cdots, \beta_i)=0$ for $i=1, 2, \cdots, I$. The triangular set of polynomials TP is *square-free* (respectively, is *irreducible*) if $P_i(\beta_1, \beta_2, \cdots, \beta_{i-1}, X_i)$ is a square-free (respectively, an irreducible) polynomial in $\mathbf{L}(\beta_1, \beta_2, \cdots, \beta_{i-1})[X_i]$ for every solution $S=\{\beta_1, \beta_2, \cdots, \beta_I\}$ of TP. The number of solutions of a square-free triangular set of polynomials TP is equal to its degree $d(TP)$. We denote by $\mathbf{L}(S)$ the subfield of **K** generated by S and **L** for each solution $S=\{\beta_1, \beta_2, \cdots, \beta_I\}$ of TP, and π_S the natural projection of $\mathbf{L}\langle TP\rangle$ onto $\mathbf{L}(S)$ defined by $\pi_S(x_i)=\beta_i$. We also denote by π_S the projection of $\mathbf{L}\langle TP\rangle((t))$ onto $\mathbf{L}(S)((t))$ defined by $\pi_S(\sum_k u_k t^k)=\sum_k \pi_S(u_k) t^k$.

Now, let us consider a place $\alpha \in \boldsymbol{K} \cup \{\infty\}$ of $\hat{C}_0$. A *system of rational Puiseux expansions* of $\hat{C}$ above α (cf. [Du3]) is defined as a set

$$R=\{(TP_j, \varphi_j(t), \psi_j(t))\}_{1\leqq j\leqq J}$$

where

— TP_j is a square-free triangular set of polynomials over $\mathbf{K}_0(\alpha)$.

— $\varphi_j(t)$ and $\psi_j(t)$ are power series in $\mathbf{K}_0(\alpha)\langle TP_j\rangle((t))$ such that $\varphi_j(t)=\alpha+\lambda_j t^{e_j}$ if $\alpha\neq\infty$ and $\varphi_j(t)=1/(\lambda_j t^{e_j})$ if $\alpha=\infty$, for some positive integer e_j and some $\lambda_j\in\mathbf{K}_0(\alpha)\langle TP_j\rangle$ such that $\pi_S(\lambda_j)\neq 0$ for every solution S of TP_j.

— The pair $\theta_{j,S}=(\pi_S(\varphi_j(t)),\pi_S(\psi_j(t)))$ is an irreducible parametrization of the curve $\hat{C}$ for every j and every solution S of TP_j.

— Every place of $\hat{C}$ above α is represented by exactly one of the parametrizations $\theta_{j,S}$.

For every S the ramification index of the place represented by $\theta_{j,S}$ is the positive integer e_j, and we have $\sum_{1\leqq j\leqq J} e_j d(TP_j)=n$.

From now on we choose one system of Puiseux expansions of $\hat{C}$ above each $\alpha\in\mathbf{K}\cap\{+\infty\}$ in order to represent the places of $\hat{C}$.

Basis of $\mathbf{L}(-\wp)$. Since $\wp$ is a place of C centered at a simple point (α,β) of C, the space $L(-\wp)$ is the set of functions $f\in\mathbf{K}(C)$ such that $f\in L(0)$ and $\tilde{f}(\alpha,\beta)=0$, where $\tilde{f}(X,Y)\in\mathbf{K}(X)[Y]$ is the representative of f of degree less than n, so that $L(-\wp)$ is the kernel of the $\mathbf{K}$-linear application $f\mapsto\tilde{f}(\alpha,\beta)$ from $L(0)$ onto $\mathbf{K}$, and the determination of a basis of $L(-\wp)$ from a basis of $L(0)$ is classical linear algebra over $\mathbf{K}$. We shall see that it is possible to choose for $L(0)$ a basis in $\mathbf{K}_0(C)$, i.e., with representatives in $\mathbf{K}_0(X)[Y]$. Then the basis for $L(-\wp)$ is in $\mathbf{K}_1(C)$, as required for Algorithm III in § 1.

Basis of $L(0)$. The computation of a basis for $L(0)$ is not so simple. Let A denote, as above, the ring of integral elements of $\mathbf{K}(C)$. It is a free module of rank n over $\mathbf{K}[X]$, and a function f on C is in A if and only if it has positive order at every place $\wp$ of C which is not above ∞. Let $\nu_f=\min_{\wp|\infty}(w_\wp(f))$. Then a function f on C is in $L(0)$ if and only if it is in A and satisfies $\nu_f\geqq 0$.

DEFINITION. Here a *basis of A above* $\mathbf{K}_0$ is a basis of A (considered as a free module over $\mathbf{K}[X]$) which is contained in $\mathbf{K}_0(C)$. A *free family of A above* $\mathbf{K}_0$ is a set of n elements of A linearly independent over $\mathbf{K}[X]$ and contained in $\mathbf{K}_0(C)$.

The idea of the algorithm is to build a free family $E=\{f_1,f_1,\cdots,f_n\}$ of A above $\mathbf{K}_0$("INITIALIZE" part of the algorithm below), and to modify it progressively until it becomes a basis of A above $\mathbf{K}_0$ ("NORMALIZE"). This basis in turn is progressively modified until it becomes "normal at infinity" ("NORMALIZE AT INFINITY"). From such a basis is easily derived a basis of $L(0)$ over $\mathbf{K}$ which is contained in $\mathbf{K}_0(C)$ ("CONCLUSION"). In addition, there is a "REDUCE" part, where we describe a subalgorithm used in the "NORMALIZE" part of the algorithm.

INITIALIZE. Let $g_i(X)=a_n(X)^i$ for $i=1,2,\cdots,n-1$ and

$$E=\{1,g_1(X)y,g_2(X)y^2,\cdots,g_{n-1}(X)y^{n-1}\}.$$

Then E is a free family of A above $\mathbf{K}_0$.

Indeed $g_i(X)y^i$ is integral because $a_n(X)y$ is integral (cf. § 1). A better choice for the $g_i(X)$'s is described in the remark at the end of this section. Note that each root of $g_i(X)$ is a root of $a_n(X)$ and consequently of $D(X)$ (as usual, $D(X)$ denotes the discriminant of $F(X,Y)$ considered as a polynomial in Y).

NORMALIZE. Progressively modify E in $\mathbf{K}_0(C)$ to get a basis of A above $\mathbf{K}_0$. This is the *normalization* of C, and the most difficult part of the algorithm. We describe here a "rational" version of the Dedekind-Weber method (cf. [Du2] for details). An alternative would be the normalization algorithm by Trager [Trag2], which is derived from an algorithm by Zassenhaus and Ford [Fo] for number fields.

Let $\alpha \in \mathbf{K}$ be any root of $D(X)$, and compute the rational Puiseux expansions of the curve $\hat{C}$ above α. As usual with dynamic evaluation, the result is a set $\{R_k, D_k(x)\}_{1 \leqq k \leqq K}$ where $\{D_k(X)\}_k$ is a splitting of $D(X)$, and for every root α of $D_k(X)$ the set $R_k = \{(TP_{k,j}, \varphi_{k,j}(t), \psi_{k,j}(t))\}_{1 \leqq j \leqq J_k}$ is a system of rational Puiseux expansions above α. Note that the power series $\varphi_{k,j}(t)$ are finite, and that we only need the first terms of the series $\psi_{k,j}(t)$. We certainly need their "singular part" (cf. [Du3] for a definition), and sometimes a bit more. Here it would be helpful to use "lazy evaluation" (as offered by streams in the Scheme language) for the implementation, and to look for a reasonable bound on the number of required terms in order to get an estimation of the complexity of our method.

From $\varphi_{k,j}(t) = \alpha + \lambda_{k,j} t^{e_{k,j}}$ compute the *discriminant of* A over $\mathbf{K}[X]$ by the formula

$$\operatorname{Disc}(A) = \prod_{k=1}^{K} D_k(X)^{\varepsilon_k} \in \mathbf{K}_0[X] \quad \text{where } \varepsilon_k = \sum_{j=1}^{J_k} (e_{k,j} - 1).$$

Also compute the *discriminant of* E as

$$\operatorname{Disc}(E) = \left(\prod_{i=1}^{n-1} g_i(X)^2 \right) D(X) \in \mathbf{K}_0[X].$$

It is well known that Disc (E) is a multiple of Disc (A), that they are equal (up to a multiplicative constant) if and only if E is a basis of A, and that the quotient Disc $(E)/$Disc (A) is a square in $\mathbf{K}_0[X]$. Compute $Q(X) = \operatorname{Disc}(E)/\operatorname{Disc}(A)$. Note that every root of Disc (E) or of $Q(X)$ is a root of $D(X)$, because for every $i \in \{1, 2, \cdots, n-1\}$ every root of $g_i(X)$ is a root of $D(X)$. In practice Disc (A), Disc (E), and $Q(X)$ need not be explicitly computed; they may be represented by some convenient square-free decomposition.

DEFINITION. Let $\Delta(X) \in \mathbf{K}_0[X]$. The set E is *reduced* with respect to $\Delta(X)$ if $\Delta(X)$ and Disc $(E)/$Disc (A) are coprime. A *reduction* of E with respect to $\Delta(X)$ is a free family E' of A above $\mathbf{K}_0$ such that Disc (E') divides Disc (E) and E' is reduced with respect to $\Delta(X)$.

Using the subalgorithm described below for this reduction, the algorithm now runs as follows:

For every $k \in \{1, 2, \cdots, K\}$ do
  Compute a reduction E' of E with respect to $D_k(X)$
  Replace E by E'.

The result is reduced with respect to $D(X)$, and since every root of Disc (E) is a root of $D(X)$ the result is a basis of A above $\mathbf{K}_0$, as required.

REDUCE. The reduction is a subalgorithm which is used in the normalization step described above. Starting with a free family $E = \{f_1, f_2, \cdots, f_n\}$ of A above $\mathbf{K}_0$ and a factor $\Delta(X)$ of some $D_k(X)$, it returns a reduction E' of E with respect to $\Delta(X)$. As above, $Q(X) = \operatorname{Disc}(E)/\operatorname{Disc}(A) \in \mathbf{K}_0[X]$.

We first assume that $\Delta(X)$ is irreducible in $\mathbf{K}_0[X]$. The reduction is a succession of *reduction steps*. A reduction step starts with E and $\Delta(X)$ as above, with the additional assumption that $\Delta(X)$ divides $Q(X)$. It returns an integer $i_0 \in \{1, 2, \cdots, n\}$ and a function $g \in A \cap \mathbf{K}_0(C)$, such that replacing f_{i_0} by $g/\Delta(X)$ in E gives a free family E'

of A above $\mathbf{K}_0$ with Disc $(E') = $ Disc $(E)/\Delta(X)^2$. Before we describe the algorithm for a reduction step, let us describe the reduction algorithm:

While $\Delta(X)$ divides $Q(X)$ do
  Perform a reduction step of E with respect to $\Delta(X)$
  Let i_0 and g be the result of this reduction step
  Replace f_{i_0} by $g/\Delta(X)$ in E
  Divide $Q(X)$ by $\Delta(X)^2$

The number of reduction steps to perform is equal to one half of the largest integer μ such that $\Delta(X)^\mu$ divides $Q(X)$. Note that for any root α of $\Delta(X)$, μ is equal to the multiplicity of α as a root of $Q(X)$. Of course if $\Delta(X)$ and $Q(X)$ are coprime then $\mu = 0$ and there is no reduction step to do.

We now describe a reduction step of E with respect to $\Delta(X)$ when $\mu > 0$. The result is made of an integer $i_0 \in \{1, 2, \cdots, n\}$ and a function $g \in A \cap \mathbf{K}_0(C)$. More precisely, $g = \sum_{i=0}^n r_i(X) f_i$ where each $r_i(X)$ is a polynomial in $\mathbf{K}_0[X]$ of degree less than deg $(\Delta(X))$ and $r_{i_0}(X) = 1$.

Since $\Delta(X)$ divides precisely one $D_k(X)$, we have yet computed a system of rational Puiseux expansions

$$R = \{(TP_j, \varphi_j(t), \psi_j(t))\}_{1 \leqq j \leqq J}$$

above α for any root α of $\Delta(X)$ in $\mathbf{K}$. Now for each $i = 1, 2, \cdots, n$ and each $j = 1, 2, \cdots, J$, compute

$$f_i \circ (\varphi_j, \psi_j) = \sum_{h=0}^{+\infty} u_{i,j,h} t^h \in \mathbf{L}_j((t))$$

where $\mathbf{L}_j$ is the ring $\mathbf{K}_0(\alpha)\langle TP_j \rangle$. Note that $d_j = d(TP_j)$ and $\{b_{j,1}, b_{j,2}, \cdots, b_{j,d_j}\}$ the basis $B(\mathbf{L}_j)$ of $\mathbf{L}_j$ over $\mathbf{K}_0(\alpha)$. Also denote

$$u_{i,j,h} = \sum_{l=1}^{d_j} \gamma_{i,j,h,l} b_{j,l}.$$

Let $\mathcal{M}$ be the ordered set of (j, h, l) for $1 \leqq j \leqq J$, $1 \leqq h \leqq e_j$ and $1 \leqq l \leqq d_j$ with lexicographic order. The set $\mathcal{M}$ has n elements and the line matrix

$$\Gamma_i = (\gamma_{i,m})_{m \in \mathcal{M}}$$

has its entries in $\mathbf{K}_0(\alpha)$. Because $\Delta(X)$ divides $Q(X)$ it may be proved that the Γ_i's are linearly dependent. Compute some dependence relation $\sum_i \beta_i \Gamma_i$ between them over $\mathbf{K}_0(\alpha)$, using Gauss elimination in the square matrix with entries the $\gamma_{i,m}$'s. Choose some i_0 such that $\beta_{i_0} \neq 0$ and let $\beta_i' = \beta_i / \beta_{i_0}$. Each β_i' is equal to $r_i(\alpha)$ for a unique polynomial $r_i(X)$ in $\mathbf{K}_0[X]$ of degree less than deg $(\Delta(X))$, and clearly $r_{i_0}(X) = 1$.

We now have to prove that the function

$$\frac{g}{\Delta(X)} \quad \text{where } g = \sum_{i=1}^n r_i(X) f_i$$

is integral. It is easily seen that we have only to prove that $w_{\wp_j}(g/\Delta(X)) \geqq 0$ for $j = 1, \cdots, J$, where $\wp_j$ is the place represented by the parametrization $(\pi_S(\varphi_j), \pi_S(\psi_j))$

for any solution S of TP_j. Since $w_{\wp_j}(\Delta(X)) = e_j$ we must prove that $w_{\wp_j}(g) \geqq e_j$. But

$$g \circ (\varphi_j, \psi_j) = \sum_{i=1}^{n} r_i(\alpha + \lambda_j t^{e_j}) \sum_{h=0}^{+\infty} u_{i,j,h} t^h,$$

which is congruent modulo t^{e_j} to

$$\sum_{i=1}^{n} r_i(\alpha) \sum_{h=0}^{e_j-1} u_{i,j,h} t^h = \sum_{h=0}^{e_j-1} \left(\sum_{i=1}^{n} \beta_i' u_{i,j,h} \right) t^h$$

and the β_i''s have been chosen so that this is equal to zero. It follows that $g/\Delta(X)$ is integral, as required.

Replacing f_{i_0} with $g/\Delta(X)$ in E gives a set E' which is still in A and in $\mathbf{K}_0(C)$, and because $r_{i_0}(X) = 1$ this set E' is still a free family of A.

But usually $\Delta(X)$ is not irreducible in $\mathbf{K}_0[X]$, it is only square-free. The reduction step algorithm may be run with dynamic evaluation. The result is a set $\{g_m, \Delta_m(X)\}_{1 \leqq m \leqq M}$ where $\{\Delta_m(X)\}_{1 \leqq m \leqq M}$ is a splitting of $\Delta(X)$ and for each m we have $g_m = \sum_{i=0}^{n} r_{m,i}(X) f_i$ with $\deg(r_{m,i}(X)) < \deg(\Delta_m(X))$, $r_{m,i_m}(X) = 1$ for some i_m, and $g_m/\Delta_m(X)$ is integral. The reduction algorithm may be adapted to work with one of the pairs $(g_m, \Delta_m(X))$ instead of the pair $(g, \Delta(X))$ used in the irreducible case.

NORMALIZE AT INFINITY. Progressively modify E (which is now a basis of A above $\mathbf{K}_0$) to get a basis of A above $\mathbf{K}_0$ which is "normal at infinity," in a sense to be defined. This part used to be considered quite difficult too, and indeed here the Trager method seems to be quite inefficient [Trag2]. But using the Dedekind–Weber method and a remark in [Ka] it becomes very simple, as described below.

For this method to be simple we need the additional assumption that ∞ is a noncritical value for $F(X, Y)$, which means by definition that zero is a noncritical value for the polynomial $X^d F(1/X, Y/X)$ where d is the total degree of $F(X, Y)$. Let $a_{i,j} \in \mathbf{K}_0$ be the coefficient of $X^j Y^i$ in $F(X, Y)$ for every (i, j). Define $\xi(T) \in \mathbf{K}_0[T]$ as

$$\xi(T) = \sum_{i=0}^{d} a_{i,d-i} T^i.$$

It is easy to see that ∞ is noncritical for $F(X, Y)$ if and only if $\xi(T)$ is square-free of degree n. If the given $F(X, Y)$ does not satisfy this assumption, then perform the following transformations on F:

— If zero is a critical value for $F(X, Y)$, then find some $\alpha \in \mathbf{K}_0$ that is noncritical (cf. § 1) and replace $F(X, Y)$ by $F(X + \alpha, Y)$.

— Replace $F(X, Y)$ by $X^d F(1/X, Y/X)$.

The resulting polynomial $F(X, Y)$ is still primitive in Y. Note that in Algorithm III we could as well assume, with minor modifications, that $F(X, Y)$ is monic in Y (instead of primitive). It would simplify the initialization step above since we could take $g_i(X) = 1$ for every i, but it would make it more difficult to assume that ∞ is noncritical. It would oblige to replace $F(X, Y)$ by some polynomial with the same total degree d, but with degree in Y also equal to d (instead of n). Since the complexity of the algorithm seems to depend more on n than on d we prefer to choose the primitiveness assumption.

Now each place above ∞ on $\hat{C}$ corresponds to a simple point on $\hat{C}$, and to a parametrization of $\hat{C}$ of the form

$$\left(\varphi_\beta(t)=\frac{1}{t},\ \psi_\beta(t)=\frac{\beta}{t}+\sum_{k\geqq 0} y_{\beta,k}t^k\right)$$

where β runs among the roots of $\xi(T)$ in $\mathbf{K}$ and $y_{\beta,k}\in\mathbf{K}_1$ (here $\mathbf{K}_1=\mathbf{K}_0(\beta)$) for every $k\geqq 0$. This place is denoted $\wp_\beta$. In addition, and this is Kaltofen's remark in [Ka], the computation of the $y_{\beta,k}$'s in $\mathbf{K}_1$ can be performed without splitting, which means that the computation returns polynomials $P_k(T)$ for $k=0,1,\cdots$ in $\mathbf{K}_0[T]$, of degree less than n, such that $y_{\beta,k}=P_k(\beta)$ in the expression of y_β for every β.

From these parametrizations we may compute, for each $f\in\mathbf{K}_0(C)$, the expression of f at $\wp_\beta$:

$$f\circ(\varphi_\beta,\psi_\beta)=t^{\nu_f}\sum_{k\geqq 0} u_{f,k}(\beta)t^k$$

for some $u_{f,k}(T)\in\mathbf{K}_0[T]$ of degree less than n. Here ν_f is, as above, the minimum of the orders of f at $\wp_\beta$ for every root β of $\xi(T)$. It means that the number $u_{f,0}(\beta)$ is nonzero for at least one value of β, or equivalently that the polynomial $u_{f,0}(T)$ is not zero.

By definition, a basis $E=\{f_1,f_2,\cdots,f_n\}$ of A is *normal at infinity* if the polynomials $u_{f_i,0}(T)$ (for $1\leqq i\leqq n$) are linearly independent over $\mathbf{K}$. Note that if E is in $\mathbf{K}_0(C)$, then the $u_{f_i,0}(T)$'s are in the $\mathbf{K}_0$-vector space of polynomials in $\mathbf{K}_0[T]$ of degree less than n, which has dimension n over $\mathbf{K}_0$. So that they are independent over $\mathbf{K}$ (or equivalently over $\mathbf{K}_0$) if and only if the $n\times n$ matrix with entries in $\mathbf{K}_0$ formed by their coefficients is nonsingular. We show here how it is possible to get a basis normal at infinity from any basis of A.

If the given basis is not normal at infinity, then perform a *reduction step at infinity:* First permute the f_i's in such a way that $\nu_{f_1}\geqq\nu_{f_2}\geqq\cdots\geqq\nu_{f_n}$. From the failure of the independence test comes a nontrivial $\mathbf{K}_0$-linear relation among the $u_{f_i,0}(T)$'s, say

$$\sum_{i=1}^{n} m_i u_{f_i,0}(T)=0.$$

Let I be the largest i such that $m_i\neq 0$, let $\mu_i=\nu_{f_i}-\nu_{f_I}$ for every i, and let

$$g=\sum_{i=0}^{I} m_i X^{\mu_i}f_i\in\mathbf{K}_0(C).$$

Replace f_I by g in E, it remains a basis of A since the coefficient of f_I in g is just m_I. This is the end of the reduction step at infinity. Note that $\nu_g>\nu_{f_I}$.

If this new basis is normal at infinity then return it, and if not then perform another reduction step at infinity on it, and so on. If this algorithm does terminate, we clearly get a basis of A over $\mathbf{K}[X]$ which is normal at infinity and contained in $\mathbf{K}_0(C)$.

To prove termination, note that since $\nu_g>\nu_{f_I}$ the integer $\nu(E)$ defined as $\sum_{i=1}^n \nu_{f_i}$ increases at each reduction step at infinity. But it is bounded for the following reason: The monomials X^kf_i for $1\leqq i\leqq n$ and $0\leqq k\leqq\nu_{f_i}$ are in $L(0)$, and they are linearly independent over $\mathbf{K}$, so their number s is at most the dimension r of $L(0)$ over $\mathbf{K}$. But s is trivially equal to $\sum_{i=1}^n \max(0,\nu_{f_i}+1)$, which is greater than or equal to $\nu(E)+n$, whence $\nu(E)\leqq r-n$ and termination is proved.

CONCLUSION. Let $E=\{f_1,f_2,\cdots,f_n\}$ be the basis of A we have just obtained (normal at infinity), and define

$$B=\{X^k f_i \text{ for } 1\leqq i\leqq n \text{ and } 0\leqq k\leqq \nu_{f_i}\}.$$

Then B is a basis (in $\mathbf{K}_0(C)$) of $L(0)$ over $\mathbf{K}$.

This claim needs proof. Since E is a basis of A over $\mathbf{K}[X]$, it is clear that B is a set of $\mathbf{K}$-linearly independent elements of $L(0)$. Now using the property that E is normal at infinity we prove that B generates $L(0)$. Let $f\in L(0)$, it means that $f\in A$ and $\nu_f\geqq 0$. Since $f\in A$ there exist polynomials $M_i(X)\in\mathbf{K}[X]$ such that $f=\sum_{i=1}^n M_i(X)f_i$. Let $m_iX^{d_i}$ be the term of higher degree in $M_i(X)$ (if it is not 0), and $\mu_i=\nu_{f_i}-d_i$. Then at the place $\wp_\beta$ we have

$$(M_i(X)f_i)\circ(\varphi_\beta,\psi_\beta)=m_iu_{f_i,0}(\beta)t^{\mu_i}+\cdots.$$

Now assume that f is not in the $\mathbf{K}$-vector space generated by B, i.e., that there is some i such that $d_i>\nu_{f_i}$. Then the integer $\min_{1\leqq i\leqq n}\mu_i$ is negative. Let $\mathcal{I}$ be the set of i's such that $\mu_i=\min_{1\leqq i\leqq n}\mu_i$. Since $\nu_f\geqq 0$ we must have cancellation:

$$\sum_{i\in\mathcal{I}} m_iu_{f_i,0}(\beta)=0$$

for every β, i.e., $\sum_{i\in\mathcal{I}} m_iu_{f_i,0}(T)=0$. But this is impossible since E is normal at infinity, so B is a basis of $L(0)$ over $\mathbf{K}$, as asserted.

Remark. Here is described a better choice for the $g_i(X)$'s in the initialization above. It is better because the g_i's have the smallest possible degree, resulting in fewer reduction steps for the normalization. Let α be any root of $a_n(X)$ in $\mathbf{K}$, and compute the value $s\in\mathbf{Q}$ of the largest slope in the Newton polygon of $F(X+\alpha, Y)$ (cf., for example, [Du3] for a definition of the Newton polygon). By dynamic evaluation the result is a set of pairs $\{(s_k, a_{n,k}(X))\}_k$ with $s_k\in\mathbf{Q}$ and $\{a_{n,k}(X)\}_k$ a splitting of $a_n(X)$. Let $\sigma_{i,k}$ denote the smallest integer such that $\sigma_{i,k}\geqq is_k$. Now choose $g_i(X)=\prod_k a_{n,k}(X)^{\sigma_{i,k}}$.

We have to prove that $g_i(X)y^i$ is integer for every i between 1 and $n-1$. It means that $w_\wp(g_i(X)y^i)\geqq 0$ for any root α of $a_n(X)$, and any place $\wp$ above α. Choose such an α and such a $\wp$, and let k_0 be the value of k such that $a_{n,k_0}(\alpha)=0$. Now $w_\wp(g_i(X)y^i)=\sum_k(\sigma_{i,k}w_\wp(a_{n,k}(X)))+iw_\wp(y)$ with $w_\wp(a_{n,k}(X))=0$ for every $k\neq k_0$ and $w_\wp(a_{n,k_0}(X))=e_\wp$ because $a_{n,k_0}(X)$ is square-free. And $s_{k_0}\geqq(-w_\wp(y)/e_\wp)$ by general properties of the Newton polygon, so that $w_\wp(g_i(X)y^i)\geqq 0$ as required. In addition, note that it remains true that each root of $g_i(X)$ is a root of $a_n(X)$.

6. Conclusion. A feature of the algorithm presented here, compared to other ones, is to give the number of irreducible components first, so that it does not try to compute any factor if $F(X, Y)$ is absolutely irreducible.

Another feature is that it makes use of the space $L(0)$, which is easily computed from the normalization of the curve. Normalization in turn is not easy to compute, but it is one of the basic objects associated with the curve, so that if the absolute factorization is needed in some other algorithm (and not only for itself), the normalization may have to be computed anyway. It is what happens for example in the package for integration of algebraic functions in Scratchpad, where the computation of $L(0)$ is implemented and used to test absolute irreducibility. However, it seems that no absolute irreducibility algorithm is entirely implemented yet.

The complexity of this algorithm is essentially the complexity of the computation of the space $L(0)$, which is itself essentially the complexity of the normalization of the curve. We conjecture that this complexity is polynomial. This point would be

essentially different from the result of Lenstra, Lenstra, and Lovász [LLL] concerning rational factorization, but similar to Kaltofen's result [Ka]. A proof could run as follows. Prove that the number of elementary operations on $\mathbf{K}$ is polynomially bounded, then by a general result in dynamic evaluation it is also true for the number of elementary operations on $\mathbf{K}_0$. As for the size of the coefficients (when $\mathbf{K}_0 = \mathbf{Q}$) the best solution is probably to perform all the computations modulo some (large enough) prime number p. Such a p certainly exists, and Trager proved that "small" values of p are easily obtained [Trag3].

Actually, the algorithm described in this paper is very similar to Berlekamp algorithm for the factorization of univariate polynomials over finite fields [Be]. Both first compute the number of irreducible factors, and both by determining the dimension of a H^0-cohomology group: Galois cohomology for Berlekamp, sheaves cohomology here [Se, Chap. 2]. This comparison could prove useful if some other factorization algorithm once were derived from it. For the determination of the factors, it can be noted that the situation of absolute factorization is much better than Berlekamp's, since *any* place $\wp$ gives a factor of $F(X, Y)$.

Another point is that both Kaltofen's algorithm and the algorithm described above prove that the absolute factorization of bivariate polynomials is totally independent from rational factorization. The first to prove this result appears to have been Noether [No].

A last point is whether an absolute factorization algorithm can prove useful in practice. This question probably has no simple answer, however we may outline here some general remarks.

Absolute factorization should not be viewed as a way to replace a "big" problem (related to F) by "a lot of" (precisely r) "small" problems (related to the F_i's), for the following reasons. For simplicity, assume that F is irreducible in $\mathbf{K}_0[X, Y]$ and that F's and the F_i's are monic in Y. First, since the F_i's are conjugated over $\mathbf{K}_0$, only one of them (instead of r) has to be considered. But this F_i is not "smaller" than F: its degree is indeed smaller (its degree in Y is (n/r)), but its coefficients are in a larger field (of degree r over $\mathbf{K}_0$), so that absolute factorization replaces one problem by another one of "similar" size. That is one of the reasons why it seems impossible to give a general conclusion about the usefulness of absolute factorization, even if we forget about its cost. For example, let $F = Y^2 - 2(X+1)^2$ over $\mathbf{K}_0 = \mathbf{Q}$, and $F_i = Y - \beta(X+1)$ for a root β of $T^2 - 2$. Then F has degree two and coefficients in $\mathbf{Q}$, while F_i has degree one and coefficients in $\mathbf{Q}(\beta)$ which is a field of degree two over $\mathbf{Q}$.

Of course, one point in favour of absolute factorization is that the mathematical study of algebraic curves is easier when the curves are irreducible. Such a basic invariant of a plane curve as its genus is only defined for irreducible curves. But in practice, if we must apply to a reducible curve an algorithm which is only described for absolutely irreducible ones, we should first look more closely at the algorithm. Many of them are valid, after straightforward adaptations, with reducible curves (maybe square-free or subject to some other easy assumptions). An example is given by Newton algorithm for the computation of Puiseux expansions [Du3], or by Dedekind-Weber normalization method (described above). They are usually described for absolutely irreducible curves, while they are valid for any curve without multiple components nor "vertical" lines.

Acknowledgments. The author would like to thank the referees for their remarks on a previous version of this paper, and Erich Kaltofen, Marc Rybowicz, Barry Trager, and Carlo Traverso for helpful discussions.

REFERENCES

[BCGW] C. BAJAJ, J. CANNY, T. GARRITY, AND J. WARREN, *Factoring rational polynomials over the complexes*, in Proc. Internat. Symposium on Symbolic and Algebraic Computation, 1989, ACM Press, New York, 1989, pp. 81-90.

[Be] E. R. BERLEKAMP, *Factoring polynomials over finite fields*, Bell Systems Tech. J., 46 (1967), pp. 1853-1859.

[Bl] G. A. BLISS, *Algebraic functions*, Ann. Math. Soc. Colloquium Publ., Vol. 16, 1933.

[BN] A. W. VON BRILL AND M. NOETHER, *Die Entwicklung der Theorie der algebraischen Funktionen in alter und neuerer Zeit*, Jahresber. Deutsch. Math. Verein., 3 (1894), pp. 107-565.

[CG] G. CHISTOV, *Subexponential-time solving of systems of algebraic equations* I, Tech. Report E-9-83, Steklov Math. Institute, Leningrad, USSR, 1983.

[Co] J. COATES, *Construction of rational functions on a curve*, Proc. Cambridge Philos. Soc., 68 (1968), pp. 105-123.

[Da] J. H. DAVENPORT, *On the integration of algebraic functions*, Lecture Notes in Computer Science 102, Springer-Verlag, Berlin, New York, 1981.

[DW] R. DEDEKIND AND H. WEBER, *Theorie der algebraischen Funktionen einer Veränderlichen*, J. Reine Angew. Math., 92 (1882), pp. 181-290.

[DD] C. DICRESCENZO AND D. DUVAL. *Algebraic Extensions and Algebraic Closure in Scratchpad*, Symbolic and Algebraic Computation (ISSAC'88), Lecture Notes in Computer Science 358, Springer-Verlag, Berlin, New York, 1988, pp. 440-446.

[DDD] C. DICRESCENZO, D. DUVAL, AND J. DELLA DORA, *About a New Method for Computing in Algebraic Number Fields*, EUROCAL'85, Lecture Notes in Computer Science 204, Springer-Verlag, Berlin, New York, 1985, pp. 289-290.

[Du1] D. DUVAL, *Une méthode géométrique de factorisation des polynômes en deux indéterminées*, Calsyf 3, 1983.

[Du2] ———, *Une approche géométrique de la factorisation absolue de polynômes*, Thèse d'Etat, Université de Grenoble 1, 1987, pp. 71-104.

[Du3] ———, *Rational Puiseux expansions*, Compositio Math., 70 (1989), pp. 119-154.

[Fo] D. J. FORD, *On the computation of the maximal order in a Dedekind domain*, Ph.D. thesis, Department of Mathematics, Ohio State University, Columbius, OH, 1978.

[Fu] W. FULTON, *Algebraic Curves*, W. A. Benjamin, New York, 1969.

[Go] V. G. GOPPA, *Codes and information*, Uspekhi Mat. Nauk, 39 (1984), pp. 77-120. (In Russian.) Also in Russian Math. Surveys, 39 (1984), pp. 87-141. (In English.)

[HS] J. HEINTZ AND M. SIEVEKING, *Absolute primality of polynomials is decidable in random polynomial time in the number of variables*, in Proc. 1981 Internat. Conference on Automata and Languages, Lecture Notes in Computer Science 11, Springer-Verlag, Berlin, New York, 1981, pp. 16-28.

[Ka] E. KALTOFEN, *Fast parallel absolute irreducibility testing*, J. Symb. Comput., 1 (1985), pp. 57-67.

[La] S. LANG, *Algebra*, Addison-Wesley, Reading, MA, 1965.

[LR] D. LEBRIGAND AND J.-J. RISLER, *Algorithme de Brill-Noether et codage des codes de Goppa*, Publ. du laboratoire d'analyse numérique, Université de Paris 6, Paris, France, 1986.

[LLL] A. K. LENSTRA, H. W. LENSTRA, AND L. LOVÁSZ, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 515-534.

[No] E. NOETHER, *Ein algebraisches Kriterium für absolute Irreduzibilität*, Math. Ann., 85 (1922), pp. 25-33.

[Se] J.-P. SERRE, *Groupes algébriques et corps de classes*, Hermann, Paris, 1959.

[Trag1] B. M. TRAGER, *Algebraic factoring and rational function integration*, in Proc. SYMSAC'76, 1976, pp. 219-226.

[Trag2] ———, *Integration of algebraic functions*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[Trag3] ———, *Good reduction of curves and applications*, Meeting on Computer and Commutative Algebra (COCOA II), Genova, 1989.

[Trav] C. TRAVERSO, *A study on algebraic algorithms: The normalization*, Rend. Sem. Mat. Torino, (1986), pp. 111-130.

[Wa] R. J. WALKER, *Introduction to Algebraic Curves*, Dover, New York, 1950.

[Yu] D. Y. Y. YUN, *On square-free decomposition algorithms*, in Proc. SYMSAC'76, ACM Inc., 1976, pp. 26-35.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 22-40, February 1991
© 1991 Society for Industrial and Applied Mathematics
002

DETERMINISTIC SAMPLING—A NEW TECHNIQUE FOR FAST PATTERN MATCHING*

UZI VISHKIN†

Abstract. Consider the following three-stage strategy for recognizing patterns in larger scenes:

Mimic randomization deterministically. Sample several positions of the pattern.

Search for sample. Find all occurrences of the sample in the scene.

Verify. For each occurrence of the sample, verify occurrence of the full pattern.

This strategy has led to the core of the new idea given in this paper. Consider the string matching problem. Given the pattern, a sample of its positions is carefully selected whose size is at most logarithmic (the *deterministic sample*). Then, the sample is searched for. For nonperiodic patterns, the sample has the following perhaps surprising property. It is possible to disqualify all occurrences of the sample positions but one, within each "neighborhood" of locations in the text, without any further comparisons of characters. This provides *sparse* verification.

This approach enables the text analysis (stages "search for sample" and "verify") to be performed in $O(\log^* n)$ time and optimal speedup on a PRAM. This improves on the previous fastest optimal speedup result. It also leads to a new serial algorithm for string matching that runs in linear time including preprocessing.

The approach is expected to be applicable for pragmatic pattern recognition problems.

In some sense the algorithms are based on degenerate forms of computation, such as AND and OR of a large number of bits. However, traditional machine designs do not take advantage of such degeneracies, and usual complexity measures do not even enable them to be reflected. This leads to the conclusion of the paper with some speculative thoughts on desirable capabilities that would enhance computing machinery for some pattern recognition applications.

Key words. string matching, serial algorithms, parallel algorithms, deterministic sampling

AMS(MOS) subject classifications. 68P99, 68Q20, 68T10, 68Q10

1. Introduction.

Suppose we are given a string of length n, $T[1 \cdots n]$, called the *text*, and a shorter string of length m, $P[1 \cdots m]$, called the *pattern*. The *string matching* problem is to find all "*starting*" locations $1 \leqq i \leqq n-m+1$ in the text, such that the pattern matches character by character the substring of the text $T[i, i+1, \cdots, i+m-1]$. As stated in [Ga85b], this is one of the most extensively studied problems in theoretical computer science.

The naive algorithm for the problem is as follows. Test whether each location $i = 1, 2, \cdots, n-m+1$ is a starting location by m character-by-character comparisons. This totals $O(nm)$ operations, or $O(1)$ time using nm processors on a CRCW PRAM. Nontrivial algorithms for this problem consist of two stages. In the first stage, the "*pattern analysis*," they construct a table based on analysis of the pattern only. In the second and final stage, the "*text analysis*," the text is analyzed. The table built in the first stage helps to minimize repeated reading of the same text characters.

There are several serial algorithms for the string matching problem: by Knuth, Morris, and Pratt [KMP77] (and the heuristic improvement by Boyer and Moore [BM77]), the randomized algorithm by Karp and Rabin [KR87], the real-time algorithm using a constant number of registers by Galil and Seiferas [GS83], and a serial

* Received by the editors August 30, 1989; accepted for publication (in revised form) March 23, 1990. This research was supported by National Science Foundation grants CCR-8615337 and CCR-8906949 and Office of Naval Research grant N00014-85-K-0046.

† Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742; and Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

simulation of the parallel algorithm by Vishkin [Vi85]. The first contribution concerning efficient parallel string matching was by Galil [Ga85a], where a framework benefiting from periodicity properties in strings was introduced. Similar properties were used in later parallel string matching algorithms. The algorithm in Galil's original paper runs in logarithmic time and is optimal for an alphabet whose size is fixed. Vishkin [Vi85] proposed a new idea that has led to an optimal speedup algorithm regardless of the alphabet size. A recent paper by Breslauer and Galil [BG88] added the following surprising perspective to our work. They observed that the new idea from [Vi85] implies that the string matching problem is not more difficult, from the parallel algorithmic point of view, than the problem of finding the maximum among n elements. This made possible a doubly logarithmic optimal parallel algorithm for the problem. In [KR87], Karp and Rabin present an optimal logarithmic parallel implementation of their randomized algorithm. Kendem, Landau, and Palem [KLP89] recently gave another parallel algorithm. Finally, we refer the reader to a survey on string problems by Galil [Ga85b].

Our main results include:

(1) A new linear time serial algorithm for the string matching problem.

(2) A new text analysis parallel algorithm that runs in $O(\log^* n)$ time using an optimal number of processors.

(3) The text analysis algorithm is based on a pattern analysis stage that takes $O(\log^2 m/\log\log m)$ time using an optimal number of processors.

(4) A randomized implementation of the pattern analysis needs $O(\log m)$ time, with high probability, using an optimal number of processors. Using the output of the randomized implementation, all text analysis results carry through (as deterministic results).

The deterministic sampling idea. All algorithms in the present paper rely on the following core idea. Given a nonperiodic pattern, our pattern analysis stage constructs a small "*deterministic sample* (*denoted DS*)" of pattern positions. This sample is an ordered set of size $l \leqq \log m - 1$. Specifically, $DS = [ds(1), ds(2), \cdots, ds(l)]$, where each $ds(j)$, $1 \leqq j \leqq l$, is a different integer between 1 and m. The main step of our *basic* text analysis tests whether each location $i = 1, 2, \cdots, n-m+1$ can be a starting location by l comparisons with the sample pattern positions. Some locations of the text will pass this test and some will fail, and therefore be disqualified as starting locations. A perhaps surprising property of DS implies that there is a way for drastically disqualifying at once (i.e., simultaneously, in one parallel round) additional locations in the text, so that any remaining nondisqualified location is *unique* in some successive substring of length $m/2$.

Theoretically, the deterministic sampling idea can be viewed as getting a "signature" of the pattern by using a small sample of its locations. Concise signatures are natural for randomized algorithms as shown in the algorithm of [KR87]. We selected the name *deterministic sampling* to convey the possibility of getting signatures using deterministic means. Interestingly, the Karp–Rabin signature concept does not seem to be less involved since it blends all entries of the pattern rather than samples a few positions of the pattern. Our randomized parallel version compares favorably with theirs: The pattern analysis result is logarithmic time and optimal speedup, with high probability, in both papers. However, while the Karp–Rabin text analysis result is randomized and logarithmic time (with high probability), ours is deterministic and $O(\log^* n)$ time; both results achieve optimal speedup. Randomized algorithmics, as advocated in Rabin [Ra76], is an appealing concept. Our paper follows [A78], [BR89],

[CV86], [Lu88], and [MNN89] in demonstrating another angle of this concept. The deterministic sample idea shows how a randomized way of thinking can enrich the design of deterministic algorithms.

Pattern recognition based on small samples is apparently an intuitive idea. Our contribution in this respect can be summarized as presenting the first deterministic string matching algorithms that are guided by this idea, and whose worst-case performance is provably efficient. The literature records works in this direction in the 1950s. We mention one and refer the interested reader to references therein. Suppose we are given i pattern strings and a single target string, each of length m where $i \ll m$. The problem is to find whether one of the pattern strings matches the target string. A simple observation in [Gi59] is that it is enough to read at most i positions of the target string in order to disqualify all pattern strings, but one, as possible matches for the target string. Our work relates also to the heuristic of [BM77]. They used a single "most notable" character for speeding up the algorithm of [KMP77], however, there was no guarantee that such a character would always be very helpful. Our construction can be phrased as picking a set of at most $\log m - 1$ notable characters, which is provably helpful.

Our parallel pattern analysis algorithm is slower than the one in [BG88]. However, our text analysis algorithm is faster. It is not hard to imagine instances where the pattern is available in advance and there is no pressure to process it very fast, while it is important to process the text as fast as possible. Using such justification, [U85] gave an interesting serial algorithm for an approximate string matching problem whose text analysis takes linear time, but the pattern analysis might even need exponential time. Recall that [BG88] showed that the string matching problem is not more difficult than finding the maximum among n elements. Since [Va75] showed that n processors need $\Omega(\log \log n)$ time to find the maximum among n elements on a parallel comparison model of computation, it is interesting to phrase our text analysis result as follows: assuming some preprocessing of the pattern, the text analysis problem is actually easier than finding the maximum among n elements.

There is a remarkably small number of problems for which there exist optimal parallel algorithms that run in sub-doubly-logarithmic time (i.e., $o(\log \log n)$ time). Constant time optimal parallel algorithms include: (a) OR and AND of n bits; (b) finding the minimum among n elements where the input consists of integers in the domain $[1, \cdots, n^c]$ (see [FRW88]); (c) $\log n$-coloring of a cycle, [CV86]; (d) some probabilistic computational geometric problems [St88]. A data-structure that provides for optimal $O(\log^* n)$ time and even inverse-Ackermann time parallel algorithms for some problems on trees and arrays, assuming some preprocessing, is given in [BV89]. Sub-doubly-logarithmic merging algorithms (on a CREW PRAM) were recently given in [BV90]. In [BV89], Berkman and Vishkin explain why constant-time and optimal speedup represents an ultimate theoretical goal for designers of parallel algorithms. Since for almost any interesting problem this goal is (provably) unachievable, any result that approaches this goal, such as our text analysis algorithm, is somewhat surprising.

Further applicability. We hope that the deterministic sample idea will find other applications in the pattern matching area. Our results extend to string matching in higher dimensions and approximate string matching if some assumptions are made about the pattern. The main difficulty we had in obtaining more general analytic results is that the concept of periodicity becomes vague already for two dimensions. The flow of our algorithms is quite rigid, once the deterministic sample is fixed. This invites

research for extending our algorithms to more specialized computer architectures. In the last section several alternative implementations of our ideas are considered. One of them is serial and is likely to need less than linear time in practice. The second suggests a reasonable assumption about the pattern that makes possible extension to higher dimensions. The third suggests an assumption about the pattern that makes possible extension to approximate matches. It might be relevant for some image processing applications. The fourth item is more speculative, as it suggests reconsidering some standard complexity measures under some circumstances.

The model of parallel computation that is needed for this paper is the *common concurrent-read concurrent-write* (CRCW) *parallel random access machine* (PRAM). A PRAM employs p synchronous processors all having access to a common memory. The common CRCW PRAM allows several processors to write simultaneously into the same memory location, provided that they try to write the same value. For convenience, however, we will describe our algorithms for the slightly more powerful *priority* CRCW PRAM; in case several processors attempt to write simultaneously into the same memory location, the one with the smallest index succeeds. Fortunately, all uses of the priority concurrent-write assumption are in order to solve the same problem. The next section states the problem and quotes the standard way for solving it on a common CRCW without asymptotic loss of efficiency. We comment on formulation of parallel complexity results in the present paper. While a bound of the form T time using p processors can always be stated as $O(T)$ time and (a total of) $O(pT)$ operations, the converse will also be true throughout this paper (but not in general). That is, T time and X operations will mean $O(T)$ time using X/T processors.

The paper is organized as follows. Section 3 presents the pattern analysis, and § 4 presents two basic text analyses. One runs in constant time and the other uses a linear number of operations, and thereby provides a linear time serial algorithm. Section 5 combines the two algorithms into an optimal $O(\log^* n)$ time text analysis, and § 6 concludes the paper.

For fast understanding of the main ideas, we suggest figuring out the definition of WITNESS in § 2, and the definition of the auxiliary column sample problem, its computation, and the deterministic sample in § 3. In § 4, understand the basic constant-time text analysis (including the Ricochet property). Then proceed to the basic optimal speedup algorithm. Understand how the serialization in advancing through the deterministic sample helps to reduce the total number of operations. In § 5, the main idea is in Stage 2. Understanding the input (and thereby the output) for each iteration should suffice.

2. Preliminaries.

Periodicity in strings. The main insight in Galil's [Ga85a] parallel string matching algorithm was to use the notion of periods in strings. We refer the reader to that paper for proofs of the facts stated below. Let u and w be two strings. u is a *period* of w if w is a prefix of u^k for some integer k, or equivalently if w is a prefix of uw. (This equivalence of definitions for *period* is called the *equivalence fact.*) The shortest period of a string w is the *period* of w. w is *periodic* if the length of its period is at most half its length; otherwise, it is nonperiodic.

The conflicting occurrences fact. Consider a pattern w whose period is u. Suppose w occurs at position i of some text string. Then it is impossible to have another occurrence of w at location j, for $i<j<i+|u|$.

The nonperiodic prefix fact. If the pattern is periodic (let p be the length of the period), then the prefix of the pattern of length $2p-1$ must be nonperiodic.

The fundamental observation regarding periodicity of strings, from which the above facts can be derived, is called the *gcd lemma* [LS62]: If w has two periods of length p and q, where $|w| \geqq p+q$, then w must have a period of length gcd (p, q).

Array WITNESS. Consider a nonperiodic pattern $P[1, \cdots, m]$. For any index i, $1 < i \leqq m/2$ consider laying two copies of the pattern one above the other where the first symbol of the upper copy aligns above the ith symbol of the lower copy, as in the example below, and the prefix $P[1, \cdots, m-i+1]$ of the upper copy aligns over the suffix $P[i, \cdots, m]$ of the lower copy. By the conflicting occurrences fact these prefix and suffix must be different. This means that for at least one $1 \leqq k \leqq m-i+1$, $P(k) \neq P(i-1+k)$. WITNESS (i) is one such index k (where $1 < i \leqq m/2$).

Example. Let the pattern be $P[1, \cdots, 7] = ababbaa$. This is a nonperiodic pattern and the suffix $P[3, 4, 5, 6, 7] = abbaa$ differs from the prefix $P[1, 2, 3, 4, 5] = ababb$ in the last three positions, see as below:

$$a\ b\ a\ b\ b\ a\ a$$

$$a\ b\ a\ b\ b\ a\ a$$

Therefore WITNESS (3) could be either 3 or 4 or 5, representing the "columns" in which the two copies of the pattern differ.

Comment. For the present paper we need only this definition of WITNESS. We briefly relate this definition to the discussion of the papers by [Vi85] and [BG88] in the Introduction. The new idea in [Vi85] was to use the information in array WITNESS for a very powerful mechanism (called *duel*): Suppose two candidate locations j and $j+i-1$ (in the text) whose distance i is small enough (i.e., $1 < i \leqq m/2-1$) are given. The duel mechanism enables us to eliminate at least one of these two candidates based on the contents of WITNESS (i). The reader is referred to [Vi85] for more information. Breslauer and Galil [BG88] have observed that application of the duel mechanism (as part of a string matching algorithm) and elimination of the smaller among two elements (as part of an algorithm for finding the maximum among n elements) lead to similar outcomes.

Reducing the periodic case to the nonperiodic case.

LEMMA 2.1. *Suppose we know that the pattern is periodic and can be presented as* $u^k v$, *where the string* u *is the period,* $k > 1$ *is an integer, and* v *is a proper prefix* (possibly empty) *of* u. *Let* $|u| = p$ *and suppose that all occurrence of the prefix* $P[1, \cdots, 2p-1]$ *in the text have already been found. Then all occurrences of the original pattern can be found using* $O(n)$ *additional operations and* $O(1)$ *additional time.*

Proof. The main substance of the computation below is searching for a pattern that is all ones.

Step 1. For each occurrence of $P[1, \cdots, 2p-1]$, at location i in the text, find whether it extends to an occurrence of $u^2 v$. If yes, mark bit $b_i := 1$ and otherwise mark $b_i := 0$.

Step 2. We partition the bits $b_1, \cdots, b_{n-2p-|v|+1}$ into p "strips." Strip s, $1 \leqq s \leqq p$, includes all bits whose index is $s (\text{mod } p)$ (for instance, strip 1 includes $b_1, b_{p+1}, b_{2p+1}, \cdots$).

Consider some location i in the text. The full pattern occurs at i if and only if the pattern $u^2 v$ occurs at all locations $i, i+p, \cdots, i+(k-2)p$. So, to find all occurrences of the full pattern, we simply must find every location in every strip whose bit is one and each of its successive $k-2$ bits is one. Step 3 shows how to do this for each of the strips in $O(n/p)$ operations and $O(1)$ time. Consider a strip s of length $t = n/p$.

Step 3.1. Partition the strip into $t/(k-2)$ successive subvectors of $k-2$ bits each.

Step 3.2. For each subvector find its largest and smallest zero bit in $O(k)$ operations and $O(1)$ time on a priority CRCW PRAM (which, as shown below, can be simulated on a common CRCW PRAM without asymptotic loss of efficiency).

Step 3.3. Given any bit b in the strip, the information (computed in Step 3.2 above) regarding the subvector containing the bit, as well as its successive subvector, suffices to determine in $O(1)$ operations whether all the $k-2$ successive bits of b are one.

Complexity. We have shown that any of our text analysis algorithms can be extended for the periodic case within additional $O(n)$ operations and $O(1)$ time on a common CRCW PRAM. Lemma 2.1 follows.

Common CRCW PRAM is enough. We describe our algorithms for the priority CRCW PRAM. In all kinds of instances but one, we can trivially use the common CRCW PRAM instead. Next, we characterize the one nontrivial kind of instances and show how to overcome the problem.

Consider the following problem. *Input.* Vector A of n bits. *Question.* Find the leftmost bit in A that is zero. Following [FRW88], we give an algorithm for this problem that needs $O(1)$ time and n processors on a common CRCW PRAM. (a) Partition A into $\sqrt{n}$ subvectors of length $\Theta(\sqrt{n})$ each. Using $O(n)$ operations find whether each of the subvectors has a zero bit. (b) Using $O(n)$ operations and $O(1)$ time, find the leftmost subvector containing a zero bit. For this, apply the parallel algorithm of [SV81] for finding the maximum among $m(=\sqrt{n})$ elements using m^2 processors in $O(1)$ time. (c) Apply the same algorithm for finding the leftmost zero bit in the leftmost subvector.

DEFINITION OF $\log^* n$. We denote the function symbol log by $\log^{(1)}$ and $\log^{(i)}$ is defined inductively as $\log\log^{(i-1)}$. Given a real number $r>1$, we define $\log^* r$ to be the smallest integer i such that $\log^{(i)} r \leqq 2$. It is well known that the function $\log^*$ is extremely slow in increasing and, for instance, $\log^* 2^{64000}=5$.

DEFINITION OF THE PREFIX-SUMS PROBLEM [LF-80]. *Input.* Array of n numbers $[a_1, a_2, \cdots, a_n]$. *Problem.* Find all prefix-sums $a_1+\cdots+a_i$, $1 \leqq i \leqq n$. Our parallel implementation applies parallel prefix-sums routines for the following problem. *Input.* Array of n numbers $[a_1, a_2, \cdots, a_n]$, where some of the n numbers are "marked" and the others are "unmarked." The problem is to compact all marked numbers into a shorter array. A standard technique in parallel computation (that was used in §§ 3 and 4 in [CV86], for instance) reduces this array compaction problem into the prefix-sums problem. The input for the prefix-sums problem is an array of n bits, where the value one represents a marked number and the value zero an unmarked number.

3. Pattern analysis. All algorithms in the present paper use the same *pattern analysis* stage.

Step 1. Find whether the pattern is periodic, and if yes find the period. Also compute array WITNESS.

Remark. For convenience, we assume that the pattern is nonperiodic throughout this presentation of the pattern analysis. However, if the pattern is periodic (let p be the length of the period), then by the nonperiodic prefix fact of § 2, the prefix of the pattern of length $2p-1$ must be nonperiodic. The computation of array WITNESS above, as well as the rest of the pattern analysis treats this prefix of length $2p-1$ as if it were the whole pattern.

Implementation and complexity. The pattern analysis of [Vi85] can be used for parallel computation of Step 1 in $O(\log m)$ time and $O(m)$ operations. Using the

pattern analysis from [BG88] the parallel time bound can even be improved to $O(\log\log m)$. We note that array WITNESS is needed only for the pattern analysis itself (Step 2 below) and can be deleted before proceeding to the text analysis.

The primary objective of the pattern analysis is the construction of a "*deterministic sample* (*denoted DS*)" of pattern positions. For presentation purposes we give full specification of the output of the pattern analysis only after Step 3.

We first define an auxiliary problem called *column sample.* This auxiliary problem helps us: (1) to define the deterministic sample; (2) to compute the deterministic sample. Step 2 finds a column sample. In Step 3 we derived the deterministic sample from the column sample.

Without loss of generality, suppose that m is even. Consider $m/2$ copies of the pattern $[c_1, c_2, \cdots, c_{m/2}]$ laid one on top of the other as in Fig. 3.1. The first location within copy c_2 is at the same *column* as the second location within copy c_1 and this correspondence extends to subsequent locations within c_1 and c_2. In general, the first location within copy c_i belongs to the same column as the ith location within c_1, and this correspondence extends to subsequent locations within c_1 and c_i.

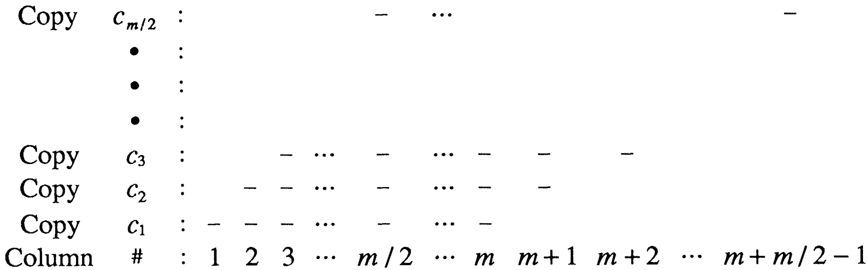


FIG. 3.1

The *column sample* problem: select at most $\log m - 1$ columns $\overline{ds}(1), \cdots, \overline{ds}(l)$, $(l < \log m)$, and associate a character car $(\overline{ds}(i))$, with each column $\overline{ds}(i)$, $1 \leqq i \leqq l$, so that the following hold.

(1) There is exactly one copy c_j for which:

(1.1) c_j intersects all these l columns (formally, $j \leqq \overline{ds}(i) < j+m$, for every $1 \leqq i \leqq l$).

(1.2) for each column $\overline{ds}(i)$, the character in c_j equals the character associated with the column (formally, $P(\overline{ds}(i)-j+1) = \text{car}\,(\overline{ds}(i))$, for every $1 \leqq i \leqq l$).

(2) For each of the other copies there is at least one column that intersects the copy and the character in the copy differs from the character associated with the column. (Formally, for each copy $c_k \neq c_j$, there is a column $\overline{ds}(i)$, $1 \leqq i \leqq l$, such that $k \leqq \overline{ds}(i) < k+m$ and $P(\overline{ds}(i)-k+1) \neq \text{car}\,(\overline{ds}(i))$).

Example. See Fig. 3.2. A nonperiodic binary pattern of length $m = 16$ is given. The suggested column sample consists of column 11 with character 1, column 12 with character 0 and column 18 with character 1. The only copy that matches these three characters (at these columns) is the seventh copy marked as c_x.

Comments. (1) Step 2 shows the existence of a solution to the column sample problem. (2) The notation $\overline{ds}$ is used for the following reason. *ds* emphasizes the strong relation that exists between the column sample and the deterministic sample—our target problem. $\overline{ds}$ suggests that the column sample is not quite the deterministic sample.

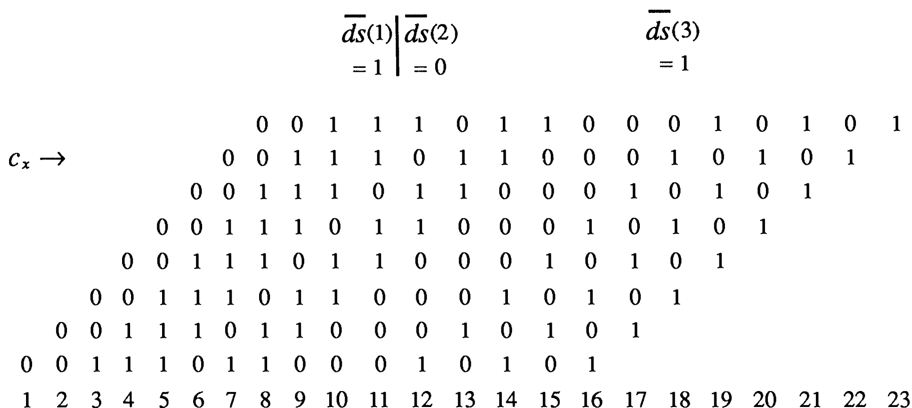


FIG. 3.2

The deterministic sample. Denote the copy that satisfies property (1) in the column sample problem by c_x. The *deterministic sample* is simply the column sample with respect to c_x. Formally, this sample is an ordered (not necessarily sorted) set $DS = [ds(1), ds(2), \cdots, ds(l)]$, of integers where $l \leqq \log m - 1$, and for each $1 \leqq j \leqq l$, $ds(j)$ is $\overline{ds}(j) - x + 1$.

Step 2. Step 2 inductively constructs sets $A_1, A_2, \cdots, A_l$, so that: (1) the base of the induction is the set $A_0 = \{c_1, c_2, \cdots, c_{m/2}\}$; (2) for each $0 \leqq i < l$, the set A_{i+1} is a nonempty subset of A_i and $|A_{i+1}| \leqq |A_i|/2$; (3) $|A_l| = 1$.

Inductive step. If A_i contains exactly one element then we set l, the cardinality of the column sample, to be i and proceed to Step 3. Otherwise, we build a nonempty subset A_{i+1} that contains at most one half of the elements in A_i. Let c_{leftmost} and $c_{\text{rightmost}}$ be the leftmost and rightmost copies in A_i, respectively. Array WITNESS will provide column $\overline{ds}(i+1)$ that contains two different characters in copies c_{leftmost} and $c_{\text{rightmost}}$. (Note that column $\overline{ds}(i+1)$ intersects every copy in A_i.) For each of these two characters find for how many copies in A_i the character in column $\overline{ds}(i+1)$ is equal to the character. Between these two characters, associate with column $\overline{ds}(i+1)$ the one with which less characters are equal. (Note that at most $|A_i|/2$ of the $|A_i|$ characters of the column will be equal to this character.) Set A_{i+1} is the subset of A_i containing all copies whose character at column $\overline{ds}(i+1)$ is equal to the selected character.

Example. Consider the construction of A_1. Note that c_1 and $c_{m/2}$, respectively, play the role of c_{leftmost} and $c_{\text{rightmost}}$, respectively. $\overline{ds}(1)$ is selected using WITNESS $(m/2)$.

Implementation and complexity. Suppose inductively that the copies belonging to A_i arrive in a compacted array (i.e., they have been renumbered from 1 to $|A_i|$). We use parallel prefix-sums for two purposes: (1) for each of the two characters of copies c_{leftmost} and $c_{\text{rightmost}}$ in column $\overline{ds}(i+1)$, finding the number of equal characters in the column (within the set A_i); and (2) to further compact the copies of A_{i+1} into an array of size $|A_{i+1}|$. Round $i+1$ takes $O(\log |A_i| / \log \log |A_i|)$ time and $O(|A_i|)$ operations using the prefix-sums algorithm of [CV89]. Since $|A_i|$ decreases geometrically, Step 2 takes a total of $O(\log^2 m / \log \log m)$ parallel time and $O(m)$ operations.

Step 3 (Deriving DS). Let c_x be the (only) element of A_l. The cardinality of DS is l and $DS = [ds(1), \cdots, ds(l)] := [\overline{ds}(1) - x + 1, \cdots, \overline{ds}(l) - x + 1]$.

Our text analyses will need the following information that was computed during the pattern analysis.

Output of the pattern analysis.

(1) The deterministic sample $DS = [ds(1), \cdots, ds(l)]$.

(2) Each set A_i, $1 \leqq i \leqq l$, in a compacted form. As indicated above, this means that the copies of A_i need to be renumbered from 1 to $|A_i|$.

Remark. Our optimal parallel algorithms use the sets A_i. The fact that the series $|A_0|$, $|A_1| \cdots |A_l|$ decreases geometrically is important for the efficiency of these algorithms.

Complexity of the pattern analysis. Since Step 2 dominates the complexity of the pattern analysis, we conclude that it needs $O(\log^2 m/\log\log m)$ time and $O(m)$ operations. The pattern analysis is given for the common CRCW PRAM. This model is used in Step 1 and in the prefix-sums computation of Step 2. A serial implementation needs $O(m)$ time.

Some practical considerations. The most important step for practical applications of the text analysis algorithms that follow is the actual set DS that is being constructed in Step 2. Particularly, we focus on the rate of reduction in the series $|A_i|$. In practice, it is most likely that the series $|A_i|$ may decrease much faster than by a factor of two. We mention in a nutshell a few common sense considerations in bringing this about. For instance, if the alphabet is of size $\sigma > 2$, we can have $|A_1|/|A_0| \leqq 1/\sigma$ by letting a character, that occurs in the pattern at most m/σ times, to guide us in the selection of a column. In general, it might be reasonable to invest more time in the pattern analysis and get a DS set that will facilitate a more efficient text analysis. For this, we may want to check all columns relative to each possible character. In each round of Step 2, we may even consider doing some backtracking (exhaustive search), where such investment makes sense. Curiously, in quite a few string matching algorithms (e.g., [Ga85a] or [W73]) the case where the alphabet is small is considered easier. The above considerations suggests that for our algorithms the opposite is correct.

Remarks. (1) Alon [A89] has constructed an example where the column sample problem needs $\Omega(\log m)$ columns. It is a nonperiodic binary sequence that is the output of a maximal linear feedback shift register.

(2) The proof of Theorem 1 in [A78] is remotely related to our deterministic sample construction. In principle, Adleman deals with a binary matrix. Looking for a small sample of columns he wants to rule out a match between a row of all zeros and any row listed in the input matrix. It is important to add that in his setting each row of the matrix is mostly ones. The crucial difference is that in our setting one of the input rows plays the role of the all zero row, and the computation needs to find such a row, since it is not known in advance which row will play this role. This row is chosen as the last survivor in the elimination process of rows according to residual minorities in columns. This explains why we feel that the deterministic sampling idea is new and only remotely related to Adleman construction.

3.1. Randomized pattern analysis. This section is not needed for understanding of the following sections. We suggest to skip it in a first reading of this manuscript.

We show how to perform the pattern analysis in $O(\log m)$ time and $O(m)$ operations, with high probability, by a randomized algorithm. The result will be a deterministic sample of size $O(\log m)$. A later comment explains why this can be guaranteed deterministically, and not only with high probability, and why all our text analysis results carry through. (Since the sample is drawn randomly, it would have been less confusing in this context to call it a *fixed*, rather than deterministic, sample.)

All our modifications refer to Step 2 above. We start Step 2, as before. We proceed, however, only until the size of the set A_i, of pattern copies, becomes at most $m/\log^2 m$. This requires $\gamma = O(\log \log m)$ rounds of the algorithm for the column sample problem.

Now, we switch to a randomized part. We outline modifications to the inductive step of Step 2. Our goal is similar. We construct smaller and smaller sets A_i. With high probability, the size of set A_{i+1} will be a constant fraction of the size of A_i. However, the difference is that we avoid performing prefix-sums, and therefore do not have compressed arrays or (deterministic) knowledge of their number of elements.

Remark. Avoiding prefix-sums computation is critical since prefix-sums need $\Omega(\log n/\log \log n)$ time using a polynomial number of processors. This was shown in [H86] together with the simulation result of [SV84], or directly in [BH87].

Finding c_{leftmost} and $c_{\text{rightmost}}$, the leftmost and rightmost copies in A_i, can be done in $O(1)$ time using $O(m/\log^2 m)$ operations on a priority CRCW PRAM. (Recall also the trick of [FRW88], as sketched in § 2, for simulation on a common CRCW PRAM.) Array WITNESS will provide the column $\overline{ds}(i+1)$, as before. The number of operations so far for each round is $O(m/\log^2 m)$ since we assign processors to jobs through the copies in A_γ.

In each round, the main effort is for selecting between two characters on column $\overline{ds}(i+1)$: either the character at copy c_{leftmost}, or the character at copy $c_{\text{rightmost}}$. We wish to select the character that is guaranteed to eliminate a constant fraction among the copies belonging to A_i with high probability. This is done in $O(1)$ time and $O(m/\log m)$ operations. The technique uses an idea from [Se89].

Overview. Let x_1 (respectively, x_2) be the number of copies in A_i whose character at column $\overline{ds}(i+1)$ is the same as at copy c_{leftmost} (respectively, $c_{\text{rightmost}}$). Note that the values of $x_1$and x_2 are unknown to us and we cannot compute them if we wish to implement each round in $O(1)$ time. Let $B[1, \cdots, (\log m)/2]$, be a vector of length $(\log m)/2$. For each integer j, $1 \leqq j \leqq (\log m)/2$, we assign the value zero to $B(j)$ with probability $x_1/(x_1+x_2)$, and the value one with probability $x_2/(x_1+x_2)$. This is done independently for different values of j, $1 \leqq j \leqq (\log m)/2$. We select for column $\overline{ds}(i+1)$ the character at copy c_{leftmost} if the total number of zeros in B is less than the total number of ones, and otherwise select the character at copy $c_{\text{rightmost}}$. This completes the overview. However, we still need to clarify several things.

(1) How to determine in $O(1)$ time whether the majority of the values in B are zero or one? For each of the $2^{(\log m)/2}$ possible binary vectors of length $(\log m)/2$, we precompute into a table the majority of zeros or ones using a total of $o(m)$ operations and $O(\log \log m)$ time. The size of the table is $2^{\log m/2}$ (which is $o(m)$). Using $(\log m)/2$ operations and $O(1)$ time per entry of the table (which is a binary vector of length $(\log m)/2$) we determine in each round whether the entry is identical with binary vector B. Determining whether vector B has more ones than zeros is done by table look-up.

(2) How to get the required probability for assignment of zero or one values to a random variable $B(j)$, $1 \leqq j \leqq (\log m)/2$? Given a random permutation of the elements in A_δ we assign processors to these elements through this permutation. Each processor standing by a copy of A_i whose character at column $\overline{ds}(i+1)$ is the same as the character at copy c_{leftmost} (respectively, $c_{\text{rightmost}}$) will try to write zero (respectively, one) at a variable $C(j)$. Since the priority CRCW PRAM is used we achieve the desired probability. Note that we will need a total of $O(\log^2 m)$ random permutations of the elements in A_δ for all rounds.

Comment. If we do not get the random permutations for free we can do the following. In [RGG89] it is shown how to generate a random permutation of n numbers in $O(\log n)$ time using $O(n \log n)$ operations on a CREW PRAM. So, had we taken

A_δ to be a set of at most $m/\log^3 m$ elements (instead of $m/\log^2 m$), we could have generated before the algorithm starts $O(\log^2 m)$ random permutations of $m/\log^3 m$ elements using a total of $O(m)$ operations and $O(\log m)$ time and have all other steps of this randomized pattern analysis carry through within the same efficiency bounds.

(3) Why our selection of the character for column $\overline{ds}(i+1)$ eliminates a constant fraction among the copies of A_i with high probability? For this we use a variant of Chernoff's bounds due to [AV79]. Each of the $y=(\log m)/2$ entries of vector B is an independent Bernoulli trial with probability of $p=x_1/(x_1+x_2)$ to get zero and $1-p$ to get one. We need only analyze cases where either p or $1-p$ are smaller than a fraction, say $f=1/10$ (since otherwise each of the two selections of a character for column $\overline{ds}(i+1)$ eliminates a fraction of at least f copies in A_i). Suppose $p<f$. We analyze the probability for getting a majority of zeros in vector B. Chernoff's bounds imply that the probability of getting at least $(1+\gamma)yp$ zeros in B is at most $\exp(-\gamma^2 yp/3)$ (exponent of the natural logarithm). We are interested in the case $\gamma=4$ and let us replace p by $f=1/10$, which is not smaller. The upper bound on the probability for getting a majority of zeros will be

$$\exp\left(-16\frac{\log m}{2}\frac{1}{10}\frac{1}{3}\right) \leqq \exp\left(-\frac{\log m}{4}\right) \leqq 2^{-(\log m)/3} = \frac{1}{m^{1/3}}.$$

With similar high probability, this process takes $O(\log m)$ rounds. Each round needs $O(1)$ time and $O(m/\log m)$ operations, totaling $O(\log m)$ time and $O(m)$ operations.

Observe that we are not yet done, since the text analysis needs to get each set A_i compressed into an array. This is achieved by means of performing a prefix-sums computation for each A_i that was obtained in the randomized part (i.e., $i>\gamma$). The main difference with respect to the deterministic Step 2 is that all these prefix-sums computations are performed *in parallel* after the entire deterministic sample and the series of A_i sets were computed. With high probability, we will have $O(\log m)$ parallel prefix-sums computations, performed in parallel. Each such computation needs $O(m/\log^2 m)$ operations and $O(\log m/\log\log m)$ time (since the assignment of processors to jobs is still through copies of A_γ). The total for the prefix-sums is $O(m/\log m)$ operations and $O(\log m/\log\log m)$ time with high probability.

Complexity. $O(\log m)$ time and $O(m)$ operations with high probability.

Comments. (1) The above algorithm is randomized. With high probability it runs in $O(\log m)$ time and $O(m)$ operations. But, what if we failed and got a sample in which $|A_{i+1}| \geqq (1-f)|A_i|$ for some i? (where f is the constant fraction that is guaranteed with high probability above.) In case this unlikely event happens, we add the following step to our randomized algorithm: run the deterministic pattern analysis. The time and number of operations bounds will remain the same, with high probability (because of the low probability of needing this additional step). An alternative to this additional step would be: repeat the randomized part until a "failure free" sample is derived.

So, obtaining a "good" sample is now guaranteed deterministically. Therefore, all our deterministic text analysis results will carry through.

(2) Yossi Matias suggested an alternative idea. Select to associate with column $\overline{ds}(i+1)$ between the character at copy c_{leftmost} and the character at copy $c_{\text{rightmost}}$ by simple coin tossing. At least one of these choices is guaranteed to eliminate one half of the copies in A_i. We can bound the probability of, say $\log m$, failures (a failure is when less than half are eliminated) in a sequence of $2\log m$ attempts by Chernoff bounds. However, what complicates (but does not make infeasible) adapting this simple

idea to our algorithms is that the cardinality of the sets A_i cannot be guaranteed to decrease geometrically at *each* round separately, with high probability.

4. Basic text analyses. We give two basic algorithms for analyzing the text: a *basic constant-time* algorithm and a *basic optimal speedup* algorithm. As implied by their names, these algorithms represent two "pure" extremes. The constant-time algorithm minimizes parallel time. It needs $O(1)$ time and $O(n \log m)$ operations. The optimal speedup algorithm minimizes the total number of operations. It needs $O(\log m)$ time and $O(n)$ operations. The next section shows how to combine ideas from both algorithms for getting $O(\log^* n)$ time and $O(n)$ operations. Unless otherwise stated, we assume that the pattern is nonperiodic. Section 2 explains how to extend any of our alternative text analyses to the periodic case. For both algorithms below we partition the first $n-m+1$ locations of the text into successive substrings of exactly $m/2$ positions each (and perhaps one substring of fewer positions). Initially, any position in the text is a *candidate* for being the start of an occurrence of the pattern. We will assume throughout that $n \geqq 3m/2$ (so that there is at least one $m/2$ block of initial candidates).

Basic constant-time text analysis.

Step 1. For each position $1 \leqq i \leqq n-m+1$ in the text, check whether the following $l=|DS|$ equalities hold: $T(i-1+ds(j))=P(ds(j))$ for every $ds(j) \in DS$. If any of these equalities does not hold, we eliminate location i as candidate.

Step 1 needs at most log m checks per any of the $n-(m-1)$ candidates, or a total of $O(n \log m)$ checks.

Next, we make a detour and present a key property of the deterministic sample. Let x be the same as in Step 3 of the pattern analysis (i.e., copy c_x is the only shift of the pattern that matches the column sample).

The Ricochet property. Let i be a candidate location in the text following Step 1. Then, based only on the candidacy of location i, we can eliminate any remaining candidate in the $x-1$ locations preceding i, as well as the $m/2-x$ locations succeeding i (that is, location $i-x+1$ through $i-1$ and $i+1$ through $i+m/2-x$).

The word "ricochet" is meant to convey the following. A candidate location is determined using matches with the deterministic sample (that consists of at most log m locations). Still this direct match of at most log m locations allows for indirect "ricochet-like" hit (or elimination of candidacy) of many (up to $m/2$ in number) locations.

Step 2. For each successive substring of length $m/2$, find its leftmost and rightmost candidates on a priority CRCW PRAM (which, in turn, can be simulated on a common CRCW PRAM without asymptotic loss of efficiency). Based on the Ricochet property, disqualify all candidates that are neither leftmost nor rightmost in their substring.

Step 2 results in having at most two candidates per any successive substring of size $m/2$. So finally, we have Step 3.

Step 3. Apply a character-by-character check to each candidate location.

Complexity of the basic constant time text analysis. $O(1)$ time using $n \log m$ processors on the common CRCW PRAM.

Basic optimal speedup (and linear serial) text analysis.

Outline. Our goal is to reduce the total number of operations from $O(n \log m)$ to $O(n)$. A first attempt at this problem is to perform Step 1 of the basic constant-time algorithm in $l=|DS|$ rounds, as follows. The input for round α is all text positions (candidates) that matched the first $\alpha-1$ positions of the deterministic sample (i.e.,

positions $ds(1), \cdots, ds(\alpha-1)$ of the pattern). In round α, check each candidate against the αth position, $ds(\alpha)$, of the deterministic sample. Unfortunately, this attempt does not lead to a bound smaller than $O(n \log m)$ on the number of operations. We overcome this problem, as follows. Each round will also include "Ricochetlike" diqualification of candidates, in the spirit of Step 2 above. This will lead to $O(n/2^{\alpha})$ candidates following round α, hence a total of $O(n)$ operations.

Step 1. For each position $1 \leqq i \leqq n-m+1$, in the text, check whether the following equality holds: $T(i-1+ds(1)) = P(ds(1))$.

In Step 2 below, we focus on a single (successive) substring of length $m/2$. All such substrings are treated similarly, and simultaneously in parallel.

Step 2.1. Find the leftmost candidate a, and rightmost candidate b in the substring. (Formally, a is the smallest index in the substring for which $T(a-1+ds(1)) = P(ds(1))$, and b is the largest such index.) Location $lg = a-1+ds(1)$ in the text is called a *left guide* and location $rg = b-1+ds(1)$ is called a *right guide.*

Consider the set of pattern positions (or shifts) $A_1 = [c_{1,1}, c_{1,2}, \cdots, c_{1,|A_1|}]$ that was obtained in the pattern analysis. We will construct two sets of text positions T_{lg} and T_{rg}. Guiding location lg induces set T_{lg} using set A_1. T_{lg} will simply be the set of $|A_1|$ text locations that align under positions of set A_1, when we align location $\overline{ds}(1)$ (this is the first location in the column sample of the pattern analysis) at the same column as lg in T. Fig. 4.1 illustrates four things: (1) the diamond-shaped structure at the top is similar to Fig. 3.1; (2) column $\overline{ds}(1)$ in the column sample and location lg in T align at the same column; (3) members of set A_1 in the pattern align at the same columns as members of the set T_{lg}; (4) location a in T is a member of T_{lg}. (Location a in T and column $c_{1,z}$ must align at the same column, for some $c_{1,z}$ that is a member of set A_1.)

Similarly, guiding location rg induces set T_{rg} using set A_1. T_{rg} is the set of text locations that are aligned with set A_1 when location $\overline{ds}(1)$ in the diamond shape is aligned at the same column as location rg in T.

The key correctness observation. Consider a location in the substring of the text. If it is neither in set T_{lg} nor in set T_{rg}, then it cannot be a start of an occurrence.

Proof. The observation follows from the following facts: (1) $b-a<m/2$. (2) Occurrence of the pattern can be in one of the $x-1$ text locations preceding b, only if the text location is in T_{rg}. (3) Occurrence can be in one of the $m/2-x$ text locations succeeding a, only if the text location is in T_{lg}. (4) There is no occurrence in locations of the substring that precede a or succeed b (by the selection of a and b).

Throughout the algorithm we mark as noncandidates locations of the text for which our computation indicates that occurrence is impossible (e.g., in Step 1 above). A possibly confusing fact is that sets T_{lg} and T_{rg} themselves may include locations that are already noncandidates. To straighten out our terminology we refer to text locations in the T_{lg} and T_{rg} lists that are noncandidates as *straw* candidates.

Step 2.2. Using set A_1 construct the set of text positions T_{lg} (respectively, T_{rg}) that can co-exist with selecting the first entry of the column sample $\overline{ds}(1)$ aligned at the same column as location lg in T (respectively, location rg in T).

Implementation remark. Assignment of processors to jobs is always a concern in designing parallel algorithms. This concern is even more acute for algorithms whose target running time prohibit application of prefix-sums, as here: we implement each round below in constant time while prefix-sums need $\Omega(\log n/\log\log n)$ time using a

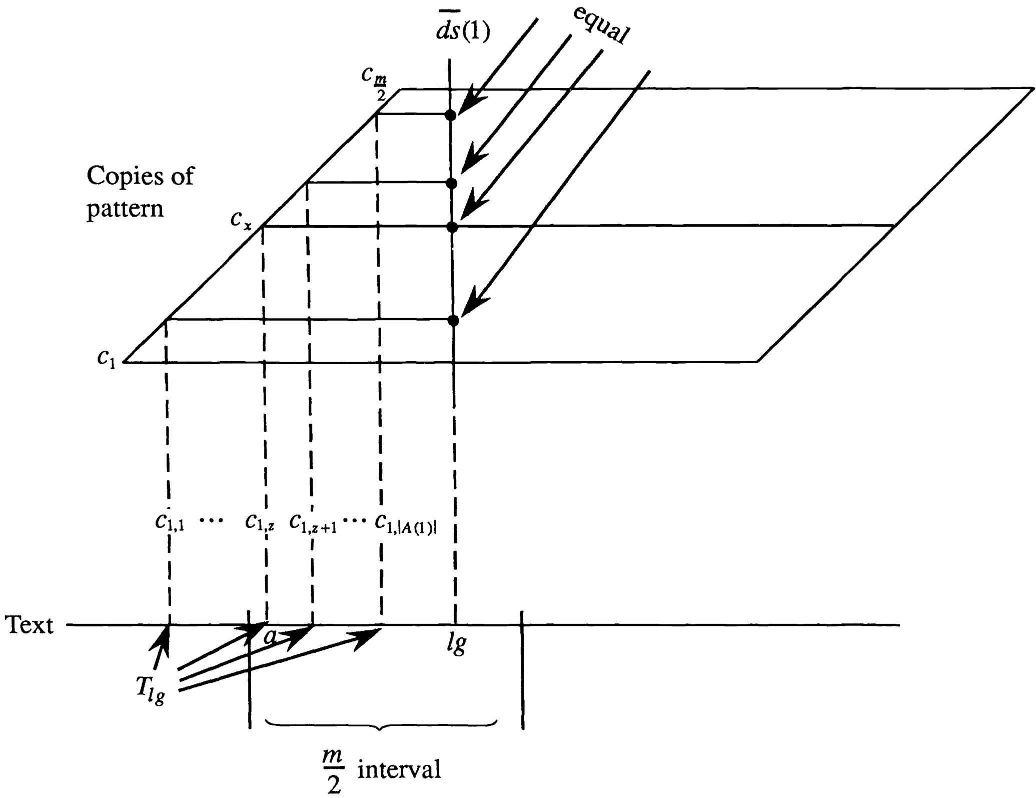


FIG. 4.1

polynomial number of processors (see references in an earlier remark). A later comment explains why it led us to include straw candidates in the T_{lg} and T_{rg} lists.

Following the above first round, Steps 1 and 2 are iterated in $l-1$ more rounds. In each round below, we focus on a single substring of length $m/2$. Other substrings are treated similarly, simultaneously in parallel. Here is an outline of round α, for $2 \leqq \alpha \leqq l$.

Step 1′. Consider every nonstraw candidate i in list T_{lg} or T_{rg} of round $\alpha-1$. For each such candidate, check whether the following equality holds: $T(i-1+ds(\sigma)) = P(ds(\alpha))$.

Step 2′.1. Find the leftmost remaining candidate, a in T, and rightmost remaining candidate, b in T, in the substring. (Formally, a is the smallest index in the substring of a candidate for which $T(a-1+ds(\alpha)) = P(ds(\alpha))$, and b is the largest such index.) Location $lg = a-1+ds(\alpha)$ in the text is called a *left guide* and location $rg = b-1+ds(\alpha)$ is called a *right guide.*

Step 2′.2. Using set A_α construct the set of text positions T_{lg} (respectively, T_{rg}) that can co-exist with selecting the αth entry of the column sample, that is column $\overline{ds}(\alpha)$, aligned at the same column as location lg in T (respectively, rg in T).

Complexity of Step 2. Since $m/2^{\alpha+1}$ is a bound on the number of elements in each T_{lg} or T_{rg} list, the bound on the total number of operations decreases by a factor of

at least two in each round. Therefore, the total number of operations is $O(n)$ and the time is $O(\log m)$. The only nontrivial detail in an exact parallel implementation of this algorithm is the issue of assignment of processors to elements of any T_{lg} and T_{rg} list in any round α.

Assignment of processors. We assign processors to the element of the T_{lg} and T_{rg} lists through the indices of the set A_α. (That is, we get every index of T_{lg} by means of adding lg to every index of A_α.) The trick is that these indices were computed in the pattern analysis (using prefix-sums). Observe that a processor will also be assigned to each straw candidate of each T_{lg} and T_{rg} list in each round. Such processor simply remains idle during the round.

The following comments shed some more light on the rounds of Step 2.

Comment 1. The key correctness observation holds also following each Step 2'.1 of each round. Specifically, each candidate at the substring of length $m/2$ must lie either in T_{lg} or T_{rg}.

Comment 2. Again, the T_{lg} and T_{rg} lists may include straw candidates. Straw candidates may come from three sources: (i) They were not in the T_{lg} or T_{rg} list for any guiding location of round $\alpha-1$. (ii) They were already straw candidates in the T_{lg} or T_{rg} list for some guiding location of round $\alpha-1$. (iii) They were candidates in the T_{lg} or T_{rg} list for some guiding location of round $\alpha-1$ but they failed the check of Step 1' in round α.

At this stage, we remain with at most $2\lceil(n-m+1)/(m/2)\rceil = O(n/m)$ candidates.

Step 3. Compare the whole pattern relative to each candidate, in a naive character-by-character manner.

Complexity of the basic optimal speedup text analysis. $O(\log m)$ time using an optimal number of processors on the common CRCW PRAM.

5. Optimal $O(\log^* n)$ time text analysis. We show how to perform the text analysis in $O(\log^* n)$ time and $O(n)$ operations. The algorithm will have three stages. The main part (Stages 1 and 2) applies the *accelerating cascades* design principle, as discussed in [CV86].

Stage 1. Run Steps 1 and 2 of the optimal speedup basic text analysis for $2\log\log^* n+2$ rounds. For this, we use the first $\delta := 2\log\log^* n+2$ positions of *DS*, the deterministic sample. The variable δ will keep track of the number of positions of *DS* that have already been "used" in Step 2 as well. The total number of elements (candidates and straw candidates) in the resulting T_{lg} and T_{rg} lists will be at most $n/(\log^* n)^2$.

Complexity. $O(n)$ operations and $O(\log\log^* n)$ (which is $o(\log^* n)$) time.

Stage 2 has $\log^* n$ iterations, each limited to constant time. The input for each iteration is a set of candidates (in T_{lg} and T_{rg} lists). As iterations proceed, the number of candidates decreases and we can apply an increasing number of tests, per each candidate at hand, in order to accelerate the candidate disqualification rate. Interestingly, while the overall serialization of events in Stages 1 and 2 together is motivated by the basic optimal speedup text analysis, each iteration of Stage 2 resembles Step 1 of the basic constant-time text analysis, where several positions from *DS* are checked at once. Stage 3 is the same as Step 3 in the basic optimal speedup text analysis.

Stage 2.

for *count* $:= \log^* n$ *downto* 1 *do*

(*Input for present iteration*: Total of at most $n/(\log^{(\text{count})} n \log^* n)$ (straw and nonstraw) candidates in the T_{lg} and T_{rg} lists.)

For each (nonstraw) candidate i, check the next $\log^{(\text{count})} n$ positions in the sample $DS=[ds(1), \cdots, ds(l)]$.
Specifically, check whether the following $\min\{l-\delta, \log^{(\text{count})} n\}$ equalities hold:
$T(i-1+ds(\delta+1))=P(ds(\delta+1))$,
$T(i-1+ds(\delta+2))=P(ds(\delta+2))$,
...
$T(i-1+ds(\min\{1, \delta+\log^{(\text{count})} n)) = P(ds(\min\{l, \delta+\log^{(\text{count})} n))$.
For each substring of length $m/2$, find its leftmost (nonstraw) candidate and its rightmost (nonstraw) candidate. Get from them the *guiding* locations and the lists T_{lg} and T_{rg}.

Since this procedure exhausts the sample DS, we end up with just one candidate in each T_{lg} and T_{rg} list.

Complexity. In iteration *count* we perform at most $O(\log^{(\text{count})} n)$ operations per each of the iteration's input candidates in $O(1)$ time. Since the number of such input candidates is bounded by $n/(\log^{(\text{count})} n \log^* n)$, we get $O(n/\log^* n)$ operations and $O(1)$ time per iteration, or a total of $O(n)$ operations and $O(\log^* n)$ time.

Stage 3. Compare the whole pattern relative to each candidate, in a naive character-by-character manner.

Comment. Actual computation of $\log^* n$. All functions used in this paper can be computed within the complexity bounds claimed here. We refer the reader to [BV89] that shows how to compute the function $\log^* n$, for instance, in constant time using n processors.

6. Further research and speculation.

(1) A possibly sublinear serial implementation. Rivest [Ri77] showed that, under some assumptions about the string matching algorithm, sublinear time cannot be achieved in the worst case, if the pattern is considerably shorter than the text. On the other hand, there were a few works whose concern was to show that some string matching algorithms need sublinear time under some assumptions about the source of the input. The difficulty about these works is that they make assumptions on what a typical input looks like. We did not find satisfactory ways for making assumptions of this kind. To demonstrate our difficulty, we show why the common probabilistic assumption that each character of the test is equally likely and that all positions are probabilistically independent does not make sense, in general. This assumption implies that if the length of the pattern is not very small relative to the length of the text, the probability of having an occurrence of the pattern is extremely small. However, in many string matching problems we have no doubt that occurrences exist and only need to find them!

Consider a serial implementation of the basic optimal speedup text analysis algorithm. We already mentioned that it runs in linear time. Still, we provide some practical ideas for enhancing its performance. Observe that if we find a match between a text character and a pattern character in Step 1 (or Step 1′) then we can immediately use it for reducing the number of candidates near this location of the text. This may save some additional comparisons between characters of the text and the pattern in the present round. In addition, recall the remark on practical considerations in § 3. The above discussion explains why, unfortunately, we do not see how to explore these ideas in a theoretically sound manner.

(2) Extension to two or higher dimensions. Suppose our pattern is a two-dimensional m-by-m array. We are not familiar with successful attempts to extend the

concept of periodicity in strings to higher dimensions. The following *conflicting occurrences assumption* resembles the case of nonperiodic patterns in strings. Consider any two positions in the text (i, j) and (i_1, j_1) that are close enough. Formally, we require that $|i - i_1| \leqq m/2$ and $|j - j_1| \leqq m/2$. Consider laying one copy of the pattern to start at (i, j) and another copy to start at (i_1, j_1). The intersection of these two copies contains (possibly several) arrays of size $m/2$ by $m/2$. The assumption about the pattern array concerns each of these $m/2$-by-$m/2$ arrays. The assumption is that the $m/2$-by-$m/2$ array must contain a position in which the two copies of the pattern have two different characters. (This resembles the information in WITNESS.) Such an assumption make it possible for our algorithmic approach to carry through efficiently.

(3) Extension to approximate matches. Consider again the two-dimensional case, where the pattern and text consist of arrays of pixels, where each pixel is characterized by its grayness (or intensity). Suppose that there are several levels of grayness. A natural concept of approximate, rather than exact, match is where only "very different" levels of grayness are defined to mismatch. (The problem is that a small difference between two levels of grayness is insufficient evidence for a mismatch.) A possible conflicting occurrences assumption will be similar to the above-suggested assumption for extensions to higher dimensions. Such an assumption would make it possible for our algorithmic approach to carry through efficiently.

(4) Speculations on complexity measures. Machine vision is one of the most frustrating application fields for any algorithm designer, for our performance as humans analyzing scenes is vastly superior to any algorithm presently imaginable for even the most powerful machines. Our algorithm may shed some light in attempting to explain this phenomenon. Power of computing machinery is often measured by number of arithmetic operations per second and other traditional computational intensity measures. Advances in computer architecture are geared to optimize such measurements and indeed computers greatly outperform humans for computationally intense tasks. On the other hand, human vision is supposedly very effective in a few very simple tasks, such as sampling a point of reference (e.g., "pick a red car in an aerial photo of a huge parking lot") and large fan-in AND (e.g., "are all cars in the parking lot red?") or OR.

We review our basic constant time text analysis. We show that it uses very degenerate computations and barely performs any "real" computation. Rather it can be implemented using only the simple tasks that humans seem to perform well. Step 1 compares characters and then takes the AND of $|DS|$ bits. (An even more "humanlike" approach would be to "associatively identify" the deterministic sample. By this we mean that given a small pattern it might be interesting to consider hypothetical computers that can retrieve, by means of a unit-cost operation, occurrences of the pattern.) Step 2 selects a leftmost (and rightmost) bit whose value is one out of each substring of $m/2$ bits. (Again, such leftmost bit can be associatively identified.) This already makes possible occurrence of the pattern very sparse. Finally, Step 3 takes the AND of m bits to verify occurrences.

This may suggest that for pattern recognition tasks, it might be less appropriate to restrict attention to conservative computational intensity measures only, but rather articulate new measures as yardsticks for novel and potent computer architectures.

Acknowledgments. Helpful discussions with Noga Alon, Amihood Amir, Gary Benson, Omer Berkman, Joseph Ja'Ja', Rao Kosaraju, Gadi Landau, Yossi Matias, Azriel Rosenfeld, and Ramakrishna Thurimella are gratefully acknowledged.

REFERENCES

[A78] L. ADLEMAN, *Two theorems on random polynomial time*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1978, pp. 75–83.

[A89] N. ALON, personal communication.

[AV79] D. ANGLUIN AND L. G. VALIANT, *Fast probabilistic algorithms for Hamiltonian circuits and matching*, J. Comput. Systems Sci., 18 (1979), pp. 155–193.

[BG88] D. BRESLAUER AND Z. GALIL, *An optimal $O(\log \log n)$ time parallel string matching algorithm*, preprint, 1988. Also appeared as a chapter in O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, *Highly-Parallelizable Problems*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 309–319.

[BH87] P. BEAME AND J. HASTAD, *Optimal bounds for decision problems on the* CRCW PRAM, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 83–93.

[BR89] B. BERGER AND J. ROMPEL, *Simulating* $(\log^2 n)$*-wise independence in* NC, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 2–7.

[BV89] O. BERKMAN AND U. VISHKIN, *Recursive *-tree parallel data-structure*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 196–203.

[BV90] ———, *On parallel integer merging*, Tech. Report UMIACS-90-15, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, January 1990.

[BM77] R. S. BOYER AND J. S. MOORE, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[CV86] R. COLE AND U. VISHKIN, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 206–219.

[CV89] ———, *Faster optimal prefix sums and list ranking*, Inform. and Comput., 81 (1989), pp. 334–352.

[FRW88] F. E. FICH, R. L. RAGDE, AND A. WIGDERSON, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[Ga85a] Z. GALIL, *Optimal parallel algorithms for string matching*, Inform. and Control, 67 (1985), pp. 144–157.

[Ga85b] ———, *Open problems in stringology*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, New York, 1985, pp. 1–8.

[Gi59] A. GILL, *Minimum-scan pattern recognition*, IRE Trans. Inform. Theory, 5 (1959), pp. 52–58.

[GS83] Z. GALIL AND J. I. SEIFERAS, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[H86] J. HASTAD, *Almost optimal lower bounds for small depth circuits*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 6–20.

[KLP89] Z. M. KEDEM, G. M. LANDAU, AND K. V. PALEM, *Optimal parallel suffix-prefix matching algorithms and applications*, in Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1989, pp. 388–398.

[KMP77] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 322–350.

[KR87] R. M. KARP AND M. O. RABIN, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., 31 (1987), pp. 249–260.

[LF80] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computations*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[LS62] R. C. LYNDON AND M. P. SCHUTZENBERGER, *The equation* $a^M = b^N c^P$ *in a free group*, Michigan Math J., 9 (1962), pp. 289–298.

[Lu88] M. LUBY, *Removing randomness in parallel computation without a processor penalty*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 162–173.

[MNN89] R. MOTWANI, J. NAOR, AND M. NAOR, *The probabilistic method yields deterministic parallel algorithms*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 8–13.

[Ra76] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21-39.

[RGG89] V. RAJAN, R. K. GHOSH, AND P. GUPTA, *An efficient parallel algorithm for random sampling*, Inform. Process. Lett., 30 (1989), pp. 265-268.

[Ri77] R. L. RIVEST, *On the worst-case behavior of string-searching algorithms*, SIAM J. Comput., 6 (1977), pp. 669-674.

[Se89] S. SEN, *Finding an approximate-median with high-probability in constant time*, manuscript, 1989.

[St88] Q. STOUT, *Constant-time geometry on* PRAM*s*, in Proc. Internat. Conference on Parallel Processing, Chicago, IL, 1988.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel model of computation*, J. Algorithms, 2 (1981), pp. 88-102.

[SV84] L. J. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409-422.

[U85] E. UKKONEN, *Finding approximate patterns in strings*, J. Algorithms, 6 (1985), pp. 132-137.

[Va75] L. G. VALIANT, *Parallelism in comparisons models*, SIAM J. Comput., 4 (1975), pp. 348-355.

[Vi85] U. VISHKIN, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91-113.

[W73] P. WEINER, *Linear pattern matching algorithm*, in Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE Computer Society, Washington, DC, 1973, pp. 1-11.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 41-55, February 1991

© 1991 Society for Industrial and Applied Mathematics

003

AMPLIFICATION OF BOUNDED DEPTH MONOTONE READ-ONCE BOOLEAN FORMULAE*

QIAN PING GU† AND AKIRA MARUOKA‡

Abstract. Let f be a Boolean function from $\{0, 1\}^n$ to $\{0, 1\}$. The amplification function A_f of f from $[0, 1]$ to $[0, 1]$ is defined as $A_f(p) = \Pr[f(\mathbf{X}_1, \cdots, \mathbf{X}_n) = 1]$, where $\mathbf{X}_1, \cdots, \mathbf{X}_n$ are independent random variables with $\Pr[\mathbf{X}_i = 1] = p$ for $1 \leqq i \leqq n$. f is said to amplify (p, q) to (p', q') if and only if $A_f(p) \leqq p'$ and $A_f(q) \geqq q'$. Let $\Sigma_d \cup \Pi_d$ be a family of monotone Boolean formulae with alternating d levels of AND gates and OR gates each having the same number of fan-ins. A Boolean formula is said to be read once when each variable in the formula occurs at most once. In this paper it is proven that the size of monotone read-once formulae in $\Sigma_d \cup \Pi_d$ that amplify $(p, p+1/m)$ to $(p', p'+1/c)$ is $\exp(\theta((d-1)(m/c)^{1/(d-1)}))$ under certain conditions.

Key words. computational complexity, Boolean function, amplification of Boolean function, monotone read-once Boolean formula

AMS(MOS) subject classification. 68C25

1. Introduction. Given a Boolean function f from $\{0, 1\}^n$ to $\{0, 1\}$, its amplification function A_f from $[0, 1]$ to $[0, 1]$ is defined as follows:

$$A_f(p) = \Pr[f(\mathbf{X}_1, \cdots, \mathbf{X}_n) = 1],$$

where $\mathbf{X}_1, \cdots, \mathbf{X}_n$ are independent random input variables with $\Pr[\mathbf{X}_i = 1] = p$ for $1 \leqq i \leqq n$. A function f is said to amplify (p, q) to (p', q'), if $A_f(p) \leqq p'$ and $A_f(q) \geqq q'$. The notion of amplification has been studied in the context of reliability of relay contact networks [6], [7] or computational complexity of certain Boolean functions such as majority functions [8]. Moore and Shannon [7] introduced the notion of amplification to design reliable relay contact networks composed of unreliable components. Using the amplification method, Valiant [8] obtained monotone formulae of size $O(n^{5.3})$ for computing the majority functions of n variables, which is the best-known upper bound of size of monotone formulae for majority. Boppana [2] showed that the amount of amplification that Valiant obtained is optimal. In particular, he showed that if monotone read-once formulae f ("read-once" means that each variable occurs at most once) amplify $(p, p+1/m)$ to $(p', p'+1/c)$, then the size of f is $\Omega((m/c)^\alpha)$, where $\alpha = \ln(2)/\ln(\sqrt{5}-1) \approx 3.3$. Ajtai and Ben-Or [1] constructed circuits of depth d and with size $\exp(O(d(m/c)^{2/d}))$ that amplify $(p, p+1/m)$ to $(p', p'+1/c)$. All the results above are obtained from the independence properties of monotone read-once formulae. Without the restriction of formulae being read-once, the known upper and lower bounds of size of formulae that amplify $(p, p+1/m)$ to $(p', p'+1/c)$ are $O((m/c)^{3.3})$ [8] and $\Omega((m/c)^2)$ [2], respectively.

In this paper we establish bounds on the size of bounded depth formulae that realize given amplification. Formulae we deal with are monotone read-once bounded depth formulae of alternating layers of OR and AND gates, where each gate in a layer has the same fan-in. Depending on the operation of the top layer (the layer farthest from the inputs), the sets of formulae with depth d are denoted Σ_d and Π_d, respectively.

* Received by the editors December 21, 1988; accepted for publication (in revised form) March 26, 1990.

† Institute of Software, Academia Sinica of China, P.O. Box 8718, Beijing 100080, China.

‡ Department of Information Engineering, Faculty of Engineering, Tohoku University, Aoba, Aramaki, Sendai 980, Japan.

The formulae used in this paper are essentially the same as the circuits used in Ajtai and Ben-Or's paper [1]. We prove that the size of the formulae in $\Sigma_d \cup \Pi_d$ that amplify $(p, p+1/m)$ to $(p', p'+1/c)$ is $\exp(\theta((d-1)(m/c)^{1/(d-1)}))$ under certain conditions. The bound is obtained from an upper bound and a lower bound. Our upper bound shows an improvement over the one obtained by Ajtai and Ben-Or [1]. The lower bound is tight to the upper bound in the sense that they differ from each other only in a multiplicative constant factor in the exponent. As a consequence of these bounds, we show that if formulae in $\Sigma_d \cup \Pi_d$ whose size is polynomial in m/c amplify $(p, p+1/m)$ to $(p', p'+1/c)$, then the depth of the formulae is $\theta(\log(m/c))$. Using the amplification method, Ajtai and Ben-Or showed that for any t with $1 \leqq t \leqq (\log n)^d$ there exist circuits of depth $2d+2$ and with size polynomial in n that compute the threshold functions TH_t, where $TH_t(x_1, \cdots, x_n) = 1$ if and only if at least t of x_i's are 1. Applying the upper bound proved in this paper, we get a better upper bound of the depth of monotone formulae of size polynomial in n that compute TH_t. We prove that for $1 \leqq t \leqq (\log n)^d$ there exist monotone formulae of depth $d+3$ and with size polynomial in n that compute TH_t. For $t = \Omega(\log n)^d$, Boppana proved that if monotone formulae of size polynomial in n compute TH_t, then the depth of the formulae is at least $d+1$ [3]. The upper bound of the depth of formulae for TH_t proved in this paper is very close to the lower bound given by Boppana.

The remainder of this paper is divided into four sections. Section 2 gives terminology we use in this paper. In §§ 3 and 4, we prove the upper bound and the lower bound of the size of formulae that amplify $(p, p+1/m)$ to $(p', p'+1/c)$, respectively. In § 5, we show how to apply our upper bound given in § 3 to find the formulae with the bound for computing Boolean threshold functions.

2. Preliminaries. A read-once formula is a formula in which each variable occurs at most once. The monotone read-once formulae we deal with in this paper are defined as follows.

For a positive integer L_1, let

$$\Sigma(L_1) = \{(x_{i_1} \vee \cdots \vee x_{i_{L_1}}) \mid x_{i_1}, \cdots, x_{i_{L_1}} \text{ are distinct variables}\},$$

$$\Pi(L_1) = \{(x_{i_1} \wedge \cdots \wedge x_{i_{L_1}}) \mid x_{i_1}, \cdots, x_{i_{L_1}} \text{ are distinct variables}\}.$$

For positive integers $L_1, \cdots, L_d (d \geqq 2)$, let

$$\Sigma(L_1, \cdots, L_d) = \{(f_1 \vee \cdots \vee f_{L_1}) \mid f_1, \cdots, f_{L_1} \text{ are in } \Pi(L_2, \cdots, L_d) \text{ and } (f_1 \vee \cdots \vee f_{L_1}) \text{ is read once}\},$$

$$\Pi(L_1, \cdots, L_d) = \{(f_1 \wedge \cdots \wedge f_{L_1}) \mid f_1, \cdots, f_{L_1} \text{ are in } \Sigma(L_2, \cdots, L_d) \text{ and } (f_1 \wedge \cdots \wedge f_{L_1}) \text{ is read once}\}.$$

For $d \geqq 1$, Σ_d and Π_d are defined as

$$\Sigma_d = \bigcup_{L_1, \cdots, L_d \geqq 1} \Sigma(L_1, \cdots, L_d)$$

and

$$\Pi_d = \bigcup_{L_1, \cdots, L_d \geqq 1} \Pi(L_1, \cdots, L_d),$$

respectively. A formula in $\Sigma_d \cup \Pi_d$ can be considered to be one consisting of alternative d layers of OR and AND gates. Note that fan-ins of the gates in the same layer are the same. The depth of a formula in $\Sigma_d \cup \Pi_d$ is defined d. Fan-in of a formula in

$\Sigma(L_1, \cdots, L_d) \cup \Pi(L_1, \cdots, L_d)$ is defined $(L_1, \cdots, L_d)$ and the size of it is defined $L_1 L_2 \cdots L_d$. The top operation of a formula in Σ_d (respectively, Π_d), i.e., the operation of the layer farthest from the inputs is defined OR (respectively, AND). The bottom operation is defined similarly. That is, we call the operation of the gates in the top (respectively, bottom) layer of a formula the top (respectively, bottom) operation of the formula. Note that since the formulae in $\Sigma(L_1, \cdots, L_d)$ (respectively, $\Pi(L_1, \cdots, L_d)$) are transformed to each other by renaming variables, the naming of variables is irrelevant in the argument to follow. So we could consider that $\Sigma(L_1, \cdots, L_d)$ (respectively, $\Pi(L_1, \cdots, L_d)$) contains essentially one formula. The formula in $\Sigma(L_1, \cdots, L_d)$ and that in $\Pi(L_1, \cdots, L_d)$ are denoted $s(L_1, \cdots, L_d)$ and $t(L_1, \cdots, L_d)$, respectively. In particular, when the top operation of it is not specified, it is denoted $u(L_1, \cdots, L_d)$.

The formula composed of $s(L'_1, \cdots, L'_k)$ (respectively, $t(L'_1, \cdots, L'_k)$) and $u(L''_1, \cdots, L''_j)$ is defined as follows. If the bottom operation of $s(L'_1, \cdots, L'_k)$ (respectively, $t(L'_1, \cdots, L'_k)$) is different from the top operation of $u(L''_1, \cdots, L''_j)$, then the composition of the two formulae is $s(L'_1, \cdots, L'_k, L''_1, \cdots, L''_j)$ (respectively, $t(L'_1, \cdots, L'_k, L''_1, \cdots, L''_j)$); otherwise, the composition is $s(L'_1, \cdots, L'_k L''_1, \cdots, L''_j)$ (respectively, $t(L'_1, \cdots, L'_k L''_1, \cdots, L''_j)$). The definition can be easily extended for the case of a formula composed of more than two formulae. It is easily seen that the effect of the composition of formulae s and u is just substituting the copies of u for every variable in s and renaming the variables of u's appropriately to make the resultant read once. Therefore, if u and u' amplify (p', q') to (p'', q'') and (p, q) to (p', q'), respectively, then the formula composed of u and u' amplifies (p, q) to (p'', q''). Since we identify a formula with the function it computes, we use the notation such as $A_{s(L_1, \cdots, L_d)}(p)$ in what follows. Since formulae in $\Sigma_d \cup \Pi_d$ are read once and monotone, the following properties of amplification function of formulae are easily seen:

$$A_{f \wedge g} = A_f A_g, \qquad A_{f \vee g} = 1 - (1 - A_f)(1 - A_g),$$

and for $0 \leqq p < p' \leqq 1$,

$$A_f(p) \leqq A_f(p'),$$

where $f \wedge g$ and $f \vee g$, as well as f and g, are assumed to be read once. These facts are easily extended to the case where fan-in of a gate is more than two.

We often use the following notation for amplification in the sequel. For any p, m, and c with $m > c > 1$ and $0 < p, p + 1/m < 1$, we say that formula f satisfies $A(p, m, c)$ if and only if

$$A_f(p + 1/m) - A_f(p) \geqq 1/c.$$

Let $P(f, p, m, c)$ denote the property that f satisfies $A(p, m, c)$. The following easily verifiable facts are also used later.

PROPOSITION 2.1. *For $n \geqq 1$, $(1 - 1/n)^n$ is a monotone increasing function in n.*

PROPOSITION 2.2. *For $n \geqq 1$, $(1 - 1/n)^n \leqq 1/e$, where e is the base of the natural logarithm.*

PROPOSITION 2.3. *For $n \geqq 1$ and $k \leqq 1$, $(1 - k/n) \geqq (1 - 1/n)^k$. For $n \geqq 1$ and $k \geqq 1$, $(1 - k/n) \leqq (1 - 1/n)^k$.*

We shall use $\log x$, $\ln x$, and $\exp(x)$ to express $\log_2 x$, $\log_e x$, and 2^x, respectively.

3. Upper bound. In this section we prove the upper bound of the size of formulae that satisfy $A(p, m, c)$.

THEOREM 3.1. *Let $0<b<\frac{1}{3}$ be fixed. There exists a constant R such that for any p, m, c and any integer d with $1/m<1/c\leqq 1-3b$, $b\leqq p$, $p+1/m\leqq 1-b$, and $d\geqq 2$ there exists a formula $f\in\Sigma_d$ (respectively, $f'\in\Pi_d$) such that*

(i) *f (respectively, f') amplifies $(p, p+1/m)$ to $(p', p'+1/c)$ for some p' with $b\leqq p'$, $p'+1/c\leqq 1-b$, and*

(ii) $\text{size}(f)\leqq \exp(R(d-1)(m/c)^{1/(d-1)})$ (*respectively*, $\text{size}(f')\leqq \exp(R(d-1)\times (m/c)^{1/(d-1)})$).

Proof. We only prove that there exists a formula $f\in\Sigma_d$ that satisfies the conditions of the theorem. For the other case, the theorem can be proved dually. The theorem is proved by induction on d.

We first prove the theorem for $d=2$. Let f denote $s(L_1, L_2)$,

$$L_2=\lceil m/(b^2c)\rceil \tag{3.1}$$

and

$$L_1=\lceil \log(1-b)/\log(1-p^{L_2})\rceil. \tag{3.2}$$

Then

$$\begin{aligned} A_f(p)&=1-(1-p^{L_2})^{L_1}\\ &\geqq 1-(1-p^{L_2})^{\log(1-b)/\log(1-p^{L_2})}=b \end{aligned} \tag{3.3}$$

and

$$\begin{aligned} A_f(p)&\leqq 1-(1-p^{L_2})^{\log(1-b)/\log(1-p^{L_2})+1}\\ &=1-(1-b)(1-p^{L_2}). \end{aligned} \tag{3.4}$$

By $L_2\geqq(1/b)^2$, $p<1-b$ and Proposition 2.2, we have

$$p^{L_2}\leqq p^{(1/b)^2}\leqq((1-b)^{1/b})^{1/b}\leqq(1/e)^{1/b}<b. \tag{3.5}$$

By (3.4), (3.5) and $1/c\leqq 1-3b$, it follows that

$$\begin{aligned} A_f(p)+1/c&\leqq 1-(1-b)(1-p^{L_2})+1/c\\ &\leqq 1-(1-b)(1-b)+1-3b\leqq 1-b. \end{aligned} \tag{3.6}$$

Put $p'=A_f(p)$. Then, by (3.3) and (3.6), we have $b\leqq p'$, $p'+1/c\leqq 1-b$. To complete the proof of (i), we need to prove

$$A_f(p+1/m)\geqq A_f(p)+1/c. \tag{3.7}$$

Since $A_f(p+1/m)\geqq 1-b$ and (3.6) imply (3.7), we can assume that $A_f(p+1/m)\leqq 1-b$. By the assumption and $A_f(p+1/m)=1-(1-(p+1/m)^{L_2})^{L_1}$, we have

$$(1-(p+1/m)^{L_2})^{L_1}\geqq b. \tag{3.8}$$

On the other hand, there exists a w with

$$p<w<p+1/m \tag{3.9}$$

such that

$$\begin{aligned} A_f(p+1/m)-A_f(p)&=(1/m)A_f'(w)\\ &=(1/m)L_1L_2(1-w^{L_2})^{L_1-1}w^{L_2-1}, \end{aligned}$$

where the prime denotes the differentiation. By (3.9), (3.8), (3.1), (3.2), and $(1-p^{L_2})^{(1/p)^{L_2}} \geqq (1-b)^{1/b}$, which is assured by (3.5) and Proposition 2.1, we have

$$\begin{aligned} A_f(p+1/m)-A_f(p) &\geqq (1/m)L_1L_2(1-w^{L_2})^{L_1}w^{L_2} \\ &\geqq (1/m)L_1L_2(1-(p+1/m)^{L_2})^{L_1}p^{L_2} \\ &\geqq (1/m)L_1L_2bp^{L_2} \\ &= (1/m)\lceil \log(1-b)/\log(1-p^{L_2})\rceil \lceil m/(b^2c)\rceil bp^{L_2} \\ &\geqq (1/(bc))(\log(1-b)/\log(1-p^{L_2})^{(1/p)^{L_2}}), \\ &\geqq (1/(bc))(\log(1-b)/\log(1-b)^{1/b}) = 1/c, \end{aligned}$$

obtaining (3.7). Thus (i) holds.

Now we estimate the size of the formula f, given by L_1L_2. Since $(1-p^{L_2})^{(1/p)^{L_2}} < e^{-1}$ by Proposition 2.2 and $\frac{2}{3} < 1-b < 1$ by the condition of the theorem, we have

$$\log(1-b)/\log(1-p^{L_2})^{(1/p)^{L_2}} \leqq 1.$$

Therefore, by $b \leqq p$,

$$\begin{aligned} \log(1-b)/\log(1-p^{L_2}) &= (1/p)^{L_2}\log(1-b)/\log(1-p^{L_2})^{(1/p)^{L_2}} \\ &\leqq (1/p)^{L_2} \leqq (1/b)^{L_2}. \end{aligned}$$

Therefore, by (3.1) and (3.2), it is easy to see that there exists a constant R such that

$$\text{size}(f) = L_1L_2 \leqq \exp(Rm/c),$$

obtaining (ii).

Assume that the theorem holds for $d \geqq 2$. We now prove the theorem for $d+1$. Let $c' = m^{1/d}c^{(d-1)/d}$. Then we have

$$m/c' = (m/c)^{(d-1)/d}, \tag{3.10}$$

$$c'/c = (m/c)^{1/d}, \tag{3.11}$$

$$1/m < 1/c' < 1/c \leqq 1-3b. \tag{3.12}$$

By (3.12), the inductive hypothesis, and $b \leqq p$, $p+1/m \leqq 1-b$, there exists a formula $f_2 \in \Pi_d$ such that f_2 amplifies $(p, p+1/m)$ to $(p'', p''+1/c')$ for some p'' with

$$b \leqq p'', \qquad p''+1/c' \leqq 1-b, \tag{3.13}$$

and by (3.10)

$$\begin{aligned} \text{size}(f_2) &\leqq \exp(R(d-1)(m/c')^{1/(d-1)}) \\ &= \exp(R(d-1)(m/c)^{1/d}). \end{aligned} \tag{3.14}$$

Note that the constant R, whose existence is guaranteed by the induction hypothesis, depends only on b. On the other hand, by (3.12), (3.13), and the induction base, there exists a formula $f_1 \in \Sigma_2$ such that f_1 amplifies $(p'', p''+1/c')$ to $(p', p'+1/c)$ for some p' with $b \leqq p'$, $p'+1/c \leqq 1-b$, and by (3.11)

$$\text{size}(f_1) \leqq \exp(Rc'/c) = \exp(R(m/c)^{1/d}). \tag{3.15}$$

Let f be the formula composed of f_1 and f_2. Then f amplifies $(p, p+1/m)$ to $(p', p'+1/c)$. On the other hand, by (3.14) and (3.15),

$$\begin{aligned} \text{size}(f) &= \text{size}(f_1)\,\text{size}(f_2) \\ &\leqq \exp(R(d-1)(m/c)^{1/d})\exp(R(m/c)^{1/d}) \\ &= \exp(Rd(m/c)^{1/d}). \end{aligned}$$

Since the type of bottom gates in f_1 is the same as that of the top gates in f_2, the depth of f is $d+1$. Hence f belongs to Σ_{d+1}. □

COROLLARY 3.2. *Let $0<b<\frac{1}{3}$ be fixed. There exists a constant R such that for any p, m and c with $1/m<1/c\leqq 1-3b$ and $b\leqq p$, $p+1/m\leqq 1-b$, there exists a formula f of depth $\lceil\log(m/c)\rceil+1$ and size $(m/c)^R$ such that $P(f, p, m, c)$ holds.*

Proof. Substituting $\lceil\log(m/c)\rceil+1$ for d in Theorem 3.1, we obtain the corollary. □

4. Lower bound. In this section we show a lower bound on the size of formulae that satisfy $A(p, m, c)$. The lower bound proved in this section is tight to the upper bound obtained in the last section in the sense that they differ only in a multiplicative constant factor in the exponent.

LEMMA 4.1. *For any u and y with $u\geqq 1$ and $0<y<1$,*

$$u(1-y)^{u-1}y\leqq 1.$$

Proof. Since the lemma holds for $u=1$, we can assume that $u>1$. $u(1-y)^{u-1}y$ can be thought of as a function of y. Let the function be denoted by $G(y)$. Then

$$G'(y)=u(1-y)^{u-1}(1-y(u-1)/(1-y)),$$

where $G'(y)$ denotes the differentiation of $G(y)$. It is easily seen that $G(y)$ takes its maximum value when $y=1/u$. Since

$$u(1-y)^{u-1}y\,|_{y=1/u}=(1-1/u)^{u-1}<1,$$

the lemma follows. □

LEMMA 4.2. *Let $0<b<\frac{1}{2}$. For any u, v, and y with $u\geqq 1$, $v\geqq 1$, $uv\geqq 2$, and $b\leqq y\leqq 1-b$,*

$$uv\geqq \exp(buv(1-y^v)^{u-1}y^{v-1}).$$

Proof. The proof is divided into two cases.

Case 1. $v\leqq \log(uv)/\log(1/y)$.

By Lemma 4.1 with y^v substituted for y, we have

$$\begin{aligned}uv(1-y^v)^{u-1}y^{v-1}&=(v/y)u(1-y^v)^{u-1}y^v\\&\leqq v/y\leqq \log(uv)/(y\log(1/y)).\end{aligned}$$

Therefore, since it is easy to see that $y\log(1/y)\geqq b$ for $b\leqq y\leqq 1-b$, we have

$$\begin{aligned}uv&\geqq \exp(y\log(1/y)uv(1-y^v)^{u-1}y^{v-1})\\&\geqq \exp(buv(1-y^v)^{u-1}y^{v-1}).\end{aligned}$$

Case 2. $v>\log(uv)/\log(1/y)$.

The condition of the case is written as

$$uvy^v<1.$$

Therefore, since $(1-y^v)^{u-1}\leqq 1$ holds for $u\geqq 1$, we have

$$uvy^v(1-y^v)^{u-1}=yuv(1-y^v)^{u-1}y^{v-1}\leqq 1.$$

Thus, by $uv\geqq 2$ and $y\geqq b$, we have

$$\begin{aligned}uv&\geqq \exp(yuv\,(1-y^v)^{u-1}y^{v-1})\\&\geqq \exp(buv(1-y^v)^{u-1}y^{v-1}).\end{aligned}$$ □

LEMMA 4.3. *Let $0<b<\frac{1}{3}$ be fixed. There exists a fixed b' with $0<b'\leqq b$ such that, if $P(t(L_1, L_2), p, m, c)$ (respectively, $P(s(L_1, L_2), p, m, c)$) holds for p, m, and c with $1/m<1/c\leqq 1-3b$ and $1-b'\leqq p$, $p+1/m<1$ (respectively, $0<p, p+1/m\leqq b'$), then there exists $s(L_1', L_2')$ (respectively, $t(L_1', L_2')$) with $L_1'L_2'\leqq L_1L_2$ such that $P(s(L_1', L_2'), p, m, c)$ (respectively, $P(t(L_1', L_2'), p, m, c)$) holds.*

Proof. We only prove the case where $P(t(L_1, L_2), p, m, c)$ is assumed. For the other case the lemma can be proved dually.

Since $P(t(L_1, 1), p, m, c)$ implies $P(s(1, L_1), p, m, c)$, the lemma holds trivially for $L_2 = 1$.

Assume that $L_2 \geqq 2$ and that $P(t(L_1, L_2), p, m, c)$ holds. We first give a lower bound on $L_1 L_2$. By $P(t(L_1, L_2), p, m, c)$, there exists a w such that

$$p < w < p + 1/m \tag{4.1}$$

and

$$\begin{aligned} A_{t(L_1,L_2)}(p+1/m) - A_{t(L_1,L_2)}(p) &= (1/m)A'_{t(L_1,L_2)}(w) \\ &= (1/m)L_1L_2(1-(1-w)^{L_2})^{L_1-1}(1-w)^{L_2-1} \\ &\geqq 1/c, \end{aligned} \tag{4.2}$$

where the prime denotes differentiation. By $L_1 \geqq 1$ and (4.2), we have

$$\begin{aligned} L_1L_2 &\geqq L_1L_2(1-(1-w)^{L_2})^{L_1-1}(1-w)^{L_2-1}/(1-w)^{L_2-1} \\ &\geqq m/(c(1-w)^{L_2-1}). \end{aligned} \tag{4.3}$$

Case 1. $p^{\lceil m/(bc) \rceil} \geqq b$.

Let $L'_1 = 1$ and $L'_2 = \lceil m/(bc) \rceil$. Then by $(p+1/m)^{L'_2} \geqq p^{L'_2}(1 + L'_2/m)$ and $p^{L'_2} \geqq b$, we have

$$\begin{aligned} A_{s(L'_1,L'_2)}(p+1/m) - A_{s(L'_1,L'_2)}(p) &= (p+1/m)^{L'_2} - p^{L'_2} \\ &\geqq p^{L'_2}(1+L'_2/m) - p^{L'_2} = L'_2 p^{L'_2}/m \\ &\geqq \lceil (m/(bc)) \rceil (b/m) \geqq 1/c. \end{aligned}$$

Take $b' = b/(1+b)$. By (4.1) and the condition of the lemma, we have $1 - w < 1 - p \leqq b' = b/(1+b)$. Therefore, by (4.3), $L_2 \geqq 2$, and $m/c > 1$,

$$\begin{aligned} L_1L_2 &\geqq m/(c(1-w)^{L_2-1}) \\ &\geqq m/(c(1-w)) \\ &> (1+b)m/(bc) \\ &> m/(bc) + 1 \geqq \lceil m/(bc) \rceil \\ &= L'_2 = L'_1 L'_2. \end{aligned}$$

Case 2. $p^{\lceil m/(bc) \rceil} < b$.

Let $L = \lfloor \log b / \log p \rfloor$. By the condition of Case 2, $L \leqq m/(bc)$. Thus if $P(t(L), p, m, c)$ holds, then taking $b' = b/(1+b)$, $L'_1 = 1$, and $L'_2 = L$, we can obtain $L_1 L_2 \geqq L'_1 L'_2$ as we did in Case 1. So we assume that $P(t(L), p, m, c)$ does not hold, i.e.,

$$(p+1/m)^L - p^L < 1/c. \tag{4.4}$$

Since $L = \lfloor \log b / \log p \rfloor$, $1 - b \leqq p$, and $b < \frac{1}{3}$,

$$\begin{aligned} b &\leqq p^L \\ &\leqq p^{(\log b/\log p)-1} = b/p \\ &\leqq b/(1-b) < 2b. \end{aligned} \tag{4.5}$$

By (4.4), (4.5), and $1/c \leqq 1 - 3b$,

$$(p+1/m)^L < 1 - b. \tag{4.6}$$

Let $p' = p^L$ and $p' + 1/m' = (p+1/m)^L$. Then by (4.4)-(4.6), we have $b \leqq p'$, $p' + 1/m' \leqq 1-b$, and $1/m' < 1/c$. Thus, by Theorem 3.1, there is a formula $s(L', L'')$ such that $P(s(L', L''), p, m', c)$ holds and $L'L'' \leqq \exp(Rm'/c)$, where R is a constant depending only on b. Let $s(L_1', L_2')$ be the formula composed of $s(L', L'')$ and $t(L)$. Then $P(s(L_1', L_2'), p, m, c)$ holds and

$$\begin{aligned} L_1'L_2' &\leqq (L) \exp(Rm'/c) \\ &\leqq (m/(bc)) \exp(Rm'/c). \end{aligned} \tag{4.7}$$

Since $(p+1/m)^L \geqq p^L(1+L/m)$ and $p^L \geqq b$,

$$1/m' = (p+1/m)^L - p^L \geqq bL/m. \tag{4.8}$$

By (4.1) and (4.6), $L \log w < \log(1-b)$, i.e.,

$$L \geqq \log(1-b)/\log w. \tag{4.9}$$

Since $1 - b' < w$ and we will fix b' such that $b' \leqq b < \frac{1}{3}$,

$$-\log w \leqq 2(1-w). \tag{4.10}$$

From (4.7)-(4.10), we obtain

$$\begin{aligned} L_1'L_2' &\leqq (m/(bc)) \exp(Rm'/c) \\ &\leqq (m/(bc)) \exp(Rm/(bcL)) \\ &\leqq (m/(bc)) \exp(Rm \log w/(bc \log(1-b))) \\ &\leqq (m/(bc)) \exp(2R(1-w)m/(-bc \log(1-b))) \\ &= \exp(\log(m/c) + \log(1/b) + R'(1-w)m/c), \end{aligned} \tag{4.11}$$

where $R' = 2R/(-b \log(1-b))$. By (4.2) and Lemma 4.1 with L_1 and $(1-w)^{L_2}$ substituted for u and y, respectively, we have

$$(1-w)m/c \leqq L_1 L_2 (1-(1-w)^{L_2})^{L_1 - 1}(1-w)^{L_2} < L_2. \tag{4.12}$$

Take $b' = b/2^{2R'}$. By (4.3), $1 - b' < w$, $L_2 \geqq 2$, (4.12), and (4.11), we have

$$\begin{aligned} L_1 L_2 &\geqq m/(c(1-w)^{L_2 - 1}) \\ &= \exp(\log(m/c) + (L_2 - 1)\log(1/(1-w))) \\ &\geqq \exp(\log(m/c) + (L_2 - 1)(\log(1/b) + 2R')) \\ &\geqq \exp(\log(m/c) + \log(1/b) + R'L_2) \\ &\geqq \exp(\log(m/c) + \log(1/b) + R'(1-w)m/c) \\ &\geqq L_1'L_2'. \end{aligned}$$

Thus, taking $b' = \min\{b/(1+b), b/2^{2R'}\}$, the result follows. □

THEOREM 4.4. *Let* $0 < b < \frac{1}{3}$ *be fixed. There exists a constant* $R > 0$ *such that for any* p, m, c *and integer* d *with* $1/m < 1/c < 1$, $b \leqq p$, $p + 1/m \leqq 1 - b$, *and* $2 \leqq d \leqq \max\{2, \ln(m/c)+1\}$, *if* $P(u(L_1, \cdots, L_d), p, m, c)$ *holds, then* $\text{size}(u(L_1, \cdots, L_d)) \geqq \exp(R(d-1)(m/c)^{1/(d-1)})$, *where* $u(L_1, \cdots, L_d)$ *denotes* $s(L_1, \cdots, L_d)$ *or* $t(L_1, \cdots, L_d)$.

Proof. We only prove the result for the case that $P(t(L_1, \cdots, L_d), p, m, c)$ is assumed. For the other case the theorem can be proved dually. Note that $P(t(L_1, \cdots, L_d), p, m, c)$ and $1/m < 1/c$ imply $L_1 \cdots L_d \geqq 2$.

Let $d=2$. For any p, m, c with $1/m<1/c<1$ and $b\leqq p$, $p+1/m\leqq 1-b$, $P(t(L_1, L_2), p, m, c)$ implies that there exists a w with $b\leqq p<w<p+1/m\leqq 1-b$ such that

$$\begin{aligned} A_{t(L_1,L_2)}(p+1/m)-A_{t(L_1,L_2)}(p) &= (1/m)A'_{t(L_1,L_2)}(w) \\ &= (1/m)L_1L_2(1-(1-w)^{L_2})^{L_1-1}(1-w)^{L_2-1} \\ &\geqq 1/c, \end{aligned} \tag{4.13}$$

where the prime denotes differentiation. Since $L_1L_2\geqq 2$, substituting L_1, L_2, and $1-w$ for u, v, and y, respectively, in Lemma 4.2 yields

$$\begin{aligned} L_1L_2 &\geqq \exp(bL_1L_2(1-(1-w)^{L_2})^{L_2-1}(1-w)^{L_2-1}) \\ &= \exp(bA'_{t(L_1,L_2)}(w)). \end{aligned} \tag{4.14}$$

By (4.13) and (4.14) we have

$$\begin{aligned} \operatorname{size}(t(L_1, L_2)) &= L_1L_2 \\ &\geqq \exp(bA'_{t(L_1,L_2)}(w)) \\ &\geqq \exp(bm/c). \end{aligned}$$

To get the theorem, we prove the following lemma.

LEMMA 4.5. *Let* $a=\min\{b'/2, 1-3b\}$. *For* $1/m<1/c\leqq a$, $2<d\leqq \ln(m/c)+1$, *and* $b\leqq p$, $p+1/m\leqq 1-b$, $P(u(L_1,\cdots,L_d), p, m, c)$ *implies*

$$\operatorname{size}(u(L_1,\cdots,L_d))\geqq \exp((b'/2)(d-1)(m/c)^{1/(d-1)}),$$

where $0<b'\leqq b$ *is as in Lemma* 4.3.

Proof. The lemma is proved by induction on d. The base of the induction, i.e., the case of $d=2$ is immediate from the argument above.

Inductive hypothesis. For $1/m<1/c\leqq a$, $2<d\leqq \ln(m/c)$, and $b\leqq p$, $p+1/m\leqq 1-b$, $P(u(L_1,\cdots,L_d), p, m, c)$ implies

$$\operatorname{size}(u(L_1,\cdots,L_d), p, m, c)\geqq \exp((b'/2)(d-1)(m/c)^{1/(d-1)}).$$

Assume for $1/m<1/c\leqq a$, $2<d\leqq \ln(m/c)$, and $b\leqq p$, $p+1/m\leqq 1-b$, that $P(t(L_1,\cdots,L_{d+1}), p, m, c)$ holds. We now show that

$$\operatorname{size}(t(L_1,\cdots,L_{d+1}))\geqq \exp((b'/2)d(m/c)^{1/d}). \tag{4.15}$$

Let

$$\begin{aligned} p_1 &= A_{s(L_2,\cdots,L_{d+1})}(p), \\ p_1+1/c_1 &= A_{s(L_2,\cdots,L_{d+1})}(p+1/m) \end{aligned}$$

and

$$\begin{aligned} p_2 &= A_{t(L_3,\cdots,L_{d+1})}(p), \\ p_2+1/c_2 &= A_{t(L_3,\cdots,L_{d+1})}(p+1/m). \end{aligned}$$

If $1/c_1\geqq 1/c$, then $P(s(L_2,\cdots,L_{d+1}), p, m, c)$ holds. By the inductive hypothesis and Appendix 1, we have

$$\begin{aligned} \operatorname{size}(t(L_1,\cdots,L_{d+1})) &\geqq \operatorname{size}(s(L_2,\cdots,L_{d+1})) \\ &\geqq \exp((b'/2)(d-1)(m/c)^{1/(d-1)}) \\ &\geqq \exp((b'/2)d(m/c)^{1/d}), \end{aligned}$$

obtaining (4.15). So we assume $1/c_1 < 1/c$. By the same reason, we can also assume that $1/c_2 < 1/c$. Note that $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ and $1/c_1 < 1/c$ imply that $L_1 \geqq 2$. The proof is now divided into several cases.

Case 1. $p_2 \geqq 1-b'$.

In this case, $1-b' \leqq p_2$, $p_2 + 1/c_2 < 1$ and $1/c_2 < 1/c \leqq a \leqq 1-3b$. By Lemma 4.3 and the fact that $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ implies $P(t(L_1, L_2), p_2, c_2, c)$, there is a formula $s(L_1', L_2')$ with $L_1'L_2' \leqq L_1L_2$ such that $P(s(L_1', L_2'), p_2, c_2, c)$ holds. By merging the bottom layer of $s(L_1', L_2')$ and the top layer of $t(L_3, \cdots, L_{d+1})$, it is easily seen that $P(s(L_1', L_2'L_3, \cdots, L_{d+1}), p, m, c)$ holds. By $L_1'L_2' \leqq L_1L_2$, the inductive hypothesis and Appendix 1,

$$\begin{aligned} \text{size}\,(t(L_1, L_2, L_3, \cdots, L_{d+1})) &\geqq L_1'L_2'L_3 \cdots L_{d+1} \\ &= \text{size}\,(s(L_1', L_2'L_3, \cdots, L_{d+1})) \\ &\geqq \exp\,((b'/2)(d-1)(m/c)^{1/(d-1)}) \\ &\geqq \exp\,((b'/2)d(m/c)^{1/d}). \end{aligned}$$

Case 2. $p_1 + 1/c_1 \leqq \frac{1}{2}$.

Since $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ implies $P(t(L_1), p_1, c_1, c)$, there exists a w with $p_1 < w < p_1 + 1/c_1$ such that

$$\begin{aligned} A_{t(L_1)}(p_1 + 1/c_1) - A_{t(L_1)}(p_1) &= (1/c_1)A'_{t(L_1)}(w) \\ &= (1/c_1)L_1 w^{L_1 - 1} \\ &\geqq 1/c, \end{aligned} \tag{4.16}$$

where the prime denotes differentiation. By $w < p_1 + 1/c_1 \leqq \frac{1}{2}$ and $L_1 \geqq 2$, $L_1 w^{L_1-1} < 1$. Therefore, by (4.16), $1/c_1 > 1/c$, contradicting the assumption of $1/c_1 < 1/c$.

Case 3. $\frac{1}{2} < p_1 + 1/c_1$ and $p_2 < 1-b'$.

Let L be an integer with $1 \leqq L \leqq L_2$. Put

$$p' = 1-(1-p_2)^L, \qquad p' + 1/c' = 1-(1-(p_2 + 1/c_2))^L.$$

If $1/c' \geqq 1/c$, then (4.15) follows by the same argument as above. So we assume that $1/c' < 1/c$ for any integer L with $1 \leqq L \leqq L_2$. We now show that there exists an integer L with $1 \leqq L \leqq L_2$ such that

$$b'/2 \leqq p', \qquad p' + 1/c' \leqq 1 - b'/2. \tag{4.17}$$

Assume that $b'/2 \leqq p_2$. In this case, let $L = 1$. Then $b'/2 \leqq p_2 = p'$. By $1/c' < 1/c \leqq a \leqq b'/2$ and $p' = p_2 < 1-b'$, we have $p' + 1/c' < 1 - b'/2$. Thus we obtain (4.17).

Now assume that $p_2 < b'/2$. In this case, let $L = \lceil \log\,(1-b'/2)/\log\,(1-p_2) \rceil$. Then we have

$$\begin{aligned} b'/2 &= 1-(1-p_2)^{\log\,(1-b'/2)/\log\,(1-p_2)} \\ &\leqq p' = 1-(1-p_2)^L \\ &\leqq 1-(1-p_2)^{\log\,(1-b'/2)/\log\,(1-p_2)+1} \\ &< 1-(1-b'/2)^2 < b'. \end{aligned} \tag{4.18}$$

By (4.18), $1/c' < 1/c \leqq b'/2$ and $b' \leqq b < \frac{1}{3}$,

$$p' + 1/c' < b' + b'/2 < \tfrac{1}{2} < 1 - b'/2. \tag{4.19}$$

Combining (4.18) and (4.19), we obtain (4.17). Moreover, by $p' + 1/c' < \frac{1}{2} < p_1 + 1/c_1$ and the fact that $1-(1-(p_2 + 1/c_2))^L$ is a monotone increasing function in L, we have $L < L_2$.

Since $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ holds, there exists a w with $p' < w < p' + 1/c'$ such that

$$\begin{aligned} A_{t(L_1,\cdots,L_{d+1})}(p+1/m) - A_{t(L_1,\cdots,L_{d+1})}(p) \\ = (1-(1-(p'+1/c'))^{L_2/L})^{L_1} - (1-(1-p')^{L_2/L_1})^{L_1} \\ = (1/c')(L_1 L_2/L)(1-(1-w)^{L_2/L})^{L_1-1}(1-w)^{L_2/L-1} \\ \geqq 1/c. \end{aligned} \tag{4.20}$$

Reasoning as we did for the base of the induction, we have, by $L_1 \geqq 2$, $L_2/L \geqq 1$, (4.17), and (4.20),

$$L_1 L_2/L \geqq \exp((b'/2)c'/c). \tag{4.21}$$

Assume that $1/c' \leqq 1/m$. By (4.21) and Appendix 1, we have

$$\begin{aligned} \operatorname{size}(t(L_1, \cdots, L_{d+1}), p, m, c) &\geqq L_1 L_2/L \\ &\geqq \exp((b'/2)c'/c) \\ &\geqq \exp((b'/2)d(m/c)^{1/d}), \end{aligned}$$

obtaining (4.15). So we can assume that $1/m < 1/c'$. By $P(s(L, L_3, \cdots, L_{d+1}), p, m, c')$ and the inductive hypothesis, we have

$$\operatorname{size}(s(L, L_3, \cdots, L_{d+1})) \geqq \exp((b'/2)(d-1)(m/c')^{1/(d-1)}). \tag{4.22}$$

By (4.21), (4.22), and Appendix 2, we have

$$\begin{aligned} \operatorname{size}(t(L_1, L_2, L_3, \cdots, L_{d+1})) &= (L_1 L_2/L)(L L_3 \cdots L_{d+1}) \\ &\geqq \exp((b'/2)c'/c)\exp((b'/2)(d-1)(m/c')^{1/(d-1)}) \\ &\geqq \exp((b'/2)d(m/c)^{1/d}). \end{aligned}$$ □

To complete the proof of the theorem, we still need to prove the statement for the case where $1/c \leqq a$ does not hold. Assume that $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ holds. If $1/m \geqq a$ we have, by $L_1 \cdots L_{d+1} \geqq 2$, $1/c < 1$, and $d \leqq \ln(m/c)$,

$$\begin{aligned} \operatorname{size}(t(L_1, \cdots, L_{d+1})) &= L_1 \cdots L_{d+1} \\ &\geqq \exp((a/a)\ln(1/a)/\ln(1/a)) \\ &\geqq \exp((a/\ln(1/a))(m/c)\ln(m/c)) \\ &\geqq \exp((a/\ln(1/a))d(m/c)^{1/d}). \end{aligned}$$

If $1/m < a < 1/c < 1$, then since $P(t(L_1, \cdots, L_{d+1}), p, m, c)$ implies $P(t(L_1, \cdots, L_{d+1}), p, m, 1/a)$, we have, by Lemma 4.5,

$$\begin{aligned} \operatorname{size}(t(L_1, \cdots, L_{d+1})) &\geqq \exp((b'/2)d(am)^{1/d}) \\ &\geqq \exp((b'/2)da^{1/d}(m/c)^{1/d}) \\ &\geqq \exp((ab'/2)d(m/c)^{1/d}). \end{aligned}$$

Thus, by taking $R = \min\{a/\ln(1/a), ab'/2\}$, the result follows. □

Remark. Since for $m \geqq 1$

$$A_{s(1,m)}(1-1/m) = (1-1/m)^m \leqq 1/e$$

and

$$A_{s(1,m)}(1-1/(2m)) = (1-1/(2m))^m \geqq \tfrac{1}{2},$$

$(1-1/m, 1-1/(2m))$ can be amplified to $(1/e, \frac{1}{2})$ by formula $s(1, m)$. Thus, we need the condition of $b \leqq p$, $p+1/m \leqq 1-b$ to obtain Theorem 4.4.

COROLLARY 4.6. *Let $0<b<\frac{1}{3}$ be fixed. If there exists a formula in $\Sigma_d \cup \Pi_d$ of size polynomial in m/c that satisfies $A(p, m, c)$ for p, m, and c with $b \leqq p$, $p+1/m \leqq 1-b$ and $m>c>1$, then $d=\Omega(\log(m/c))$.*

Proof. By Theorem 4.4, formula f stated in the corollary satisfies

$$\text{size}(f) \geqq \exp(R(d-1)(m/c)^{1/(d-1)})$$

for some constant R. On the other hand, by the condition of the corollary,

$$\text{size}(f) \leqq (m/c)^{R'}$$

for some constant R'. Thus it is easy to see that $d=\Omega(\log(m/c))$. □

5. Threshold functions. In this section we show that the upper bound proved in § 3 can be applied to show the existence of bounded depth monotone formulae that compute threshold functions. Threshold function TH_t is defined as $TH_t(x_1, \cdots, x_n)=1$ if and only if at least t of x_i's are one. The approach due to Ajtai and Ben-Or [1] and Boppana [2] is as follows. Start with small size "probabilistic" formulae to "approximate" the threshold functions. By composing independent copies of such formulae with a formula with high amplification properties, we can get much better approximation so that the resulting formulae compute the threshold functions exactly. Let $d>0$ be a fixed integer. For $1 \leqq t \leqq (\log n)^d$, Ajtai and Ben-Or [1], using the method above, have shown that there exist circuits of depth $2d+2$ and size polynomial in n that compute TH_t. Similar results are shown in [4] and [5] independently by different methods. Applying the method of Ajtai and Ben-Or to our result proved in § 3, we can get better upper bounds on the depth of monotone formulae of size polynomial in n that compute TH_t. We will show that for $1 \leqq t \leqq (\log n)^d$, there exist monotone formulae of polynomial size in n and with depth $d+3$ that compute TH_t. Boppana [3, Thm. 4.4], proved that for $t=\Omega(\log n)^d$, if some monotone formula of size polynomial in n computes TH_t, then the depth of it is at least $d+1$. The depth of the formulae constructed in this section for TH_t is very tight to the lower bound given by Boppana.

In this section, the formulae we deal with are Boolean formulae with ordinary meaning, i.e., not restricted to be in $\Sigma_d \cup \Pi_d$, unless otherwise stated. In order to get the result, we need some definitions due to Boppana [2]. Let F be a set of formulae. $\mathbf{G}$ is called a probabilistic formula over F if $\mathbf{G}$ is a random variable over F with $\Pr[\mathbf{G}=f_i]=p_i$, $f_i \in F$. If $\mathbf{G}$ is a probabilistic formula over F then $\{f_i \mid f_i \in F \text{ and } \Pr[\mathbf{G}=f_i]>0\}$ is called the support of $\mathbf{G}$. In this paper we assume that the cardinality of the support of each probabilistic formula is finite. The size of a probabilistic formula is the maximum size of any formula in its support and the depth of a probabilistic formula is the maximum depth of any formula in its support. Let A, B be subsets of $\{0, 1\}^n$ with $A \cap B=\varnothing$. A probabilistic formula $\mathbf{G}$ is a (p, q) probabilistic separator for (A, B) if for all $x \in A$ $\Pr[\mathbf{G}(x)=1] \leqq p$ and for all $x \in B$ $\Pr[\mathbf{G}(x)=1] \geqq q$. A formula f is a (p, q) distributional separator for (A, B) if $|\{x \in A \mid f(x)=1\}| \leqq p|A|$ and $|\{x \in B \mid f(x)=1\}| \geqq q|B|$.

PROPOSITION 5.1 (Boppana [2]). *Let f be a formula of s variables that amplifies (p, q) to (p', q'), and let $\mathbf{G}_1, \cdots, \mathbf{G}_s$ be an independent (p, q) probabilistic separator for (A, B). Then $f(\mathbf{G}_1, \cdots, \mathbf{G}_s)$ is a (p', q') probabilistic separator for (A, B).*

PROPOSITION 5.2 (Ajtai and Ben-Or [1]). *If $\mathbf{G}$ is a (p, q) probabilistic separator for (A, B) with $p+1-q<1/2^n$, then there is a $(0, 1)$ distributional separator f for (A, B) that is in the support of $\mathbf{G}$.*

To get the result, we also need formulae of bounded depth and polynomial size with high amplification properties.

LEMMA 5.3. *Let d be fixed positive integer. For sufficiently large n and $1 \leqq t \leqq \min\{(\log n)^d, n-(\log n)^d\}$, there is a formula $f \in \Sigma_{d+3} \cup \Pi_{d+3}$ such that f amplifies $((t-1)/n, t/n)$ to $(1/2^{n+2}, 1-1/2^{n+2})$ and* size(f) *is bounded by a polynomial in n.*

Proof. We only prove the case of d being odd and $(\log n)^d \leqq n/2$. For other cases, the lemma can be proved similarly.

Let f_3 denote $s(\lceil n/t \rceil)$. By $n/t \geqq 2$ and Proposition 2.1, we have

$$(1-t/n)^{\lceil n/t \rceil} \geqq (1-t/n)^{n/t}(1-t/n) \geqq \tfrac{1}{8}. \tag{5.1}$$

By the fact that for $1 \leqq t \leqq n/2$

$$\begin{aligned}(1-(t-1)/n)^{\lceil n/t \rceil} &= (1-t/n+1/n)^{\lceil n/t \rceil} \\ &= (1-t/n)^{\lceil n/t \rceil}(1+1/(n-t))^{\lceil n/t \rceil} \\ &\geqq (1-t/n)^{\lceil n/t \rceil}(1+1/t)\end{aligned}$$

and (5.1), we have

$$\begin{aligned}A_{f_3}(t/n) - A_{f_3}((t-1)/n) &= (1-(t-1)/n)^{\lceil n/t \rceil} - (1-t/n)^{\lceil n/t \rceil} \\ &\geqq (1-t/n)^{\lceil n/t \rceil}(1+1/t) - (1-t/n)^{\lceil n/t \rceil} \\ &= (1-t/n)^{\lceil n/t \rceil}(1/t) > 1/(8t).\end{aligned} \tag{5.2}$$

By Proposition 2.2 and (5.1), we have

$$\begin{aligned}1-1/e &\leqq 1-(1-t/n)^{n/t} \\ &\leqq 1-(1-t/n)^{\lceil n/t \rceil} \\ &= A_{f_3}(t/n) \leqq 1-\tfrac{1}{8}.\end{aligned} \tag{5.3}$$

By (5.2), (5.3), and $1/(8t) \leqq \frac{1}{8}$, f_3 amplifies $((t-1)/n, t/n)$ to $(p', p'+1/(8t))$, where $\frac{1}{8} < p'$, $p'+1/(8t) < \frac{7}{8}$. Let $b = \frac{1}{8}$ and $1/c = \frac{5}{8}$. Then $b < p'$, $p'+1/(8t) < 1-b$ and $1/(8t) < 1/c = 1-3b$. By Theorem 3.1, there is a formula $f_2 \in \Pi_{d+1}$ that amplifies $(p', p'+1/(8t))$ to $(p'', p''+1/c)$, where $b \leqq p''$, $p''+1/c \leqq 1-b$. Thus, by $1/c = \frac{5}{8}$ and $b = \frac{1}{8}$, f_2 amplifies $(p', p'+1/(8t))$ to $(\frac{1}{4}, \frac{3}{4})$. Let f_1 denote $t(L_1, L_2, L_3)$, where $L_1 = 3n$, $L_2 = 4^{L_3}$ and $L_3 = \lceil \log n/\log 3 \rceil$. By Proposition 2.1, we have for sufficiently large n

$$\begin{aligned}A_{f_1}(\tfrac{1}{4}) &= (1-(1-(\tfrac{1}{4})^{L_3})^{L_2})^{L_1} \\ &\leqq (1-(1-\tfrac{1}{4})^4)^{L_1} \\ &< (1-\tfrac{1}{4})^{3n} < 1/2^{n+2}.\end{aligned}$$

By $L_3 \geqq \log n/\log 3$ and Propositions 2.3 and 2.2, we have, for sufficiently large n,

$$\begin{aligned}A_{f_1}(\tfrac{3}{4}) &= (1-(1-(\tfrac{3}{4})^{L_3})^{L_2})^{L_1} \\ &\geqq (1-(1-3^{\log n/\log 3}/4^{L_3})^{L_2})^{L_1} \\ &= (1-(1-n/4^{L_3})^{L_2})^{L_1} \\ &\geqq (1-(1-(\tfrac{1}{4})^{L_3})^{nL_2})^{L_1} \\ &\geqq (1-(1/e)^n)^{3n} > 1-1/2^{n+2}.\end{aligned}$$

Therefore, f_1 amplifies $(\frac{1}{4}, \frac{3}{4})$ to $(1/2^{n+2}, 1-1/2^{n+2})$. Let f be composed of f_1, f_2, and f_3. Then f amplifies $((t-1)/n, t/n)$ to $(1/2^{n+2}, 1-1/2^{n+2})$ and $f \in \Pi_{d+3}$. By $t \leqq (\log n)^d$ and Theorem 3.1, we have that

$$\begin{aligned}\text{size}(f) &= \text{size}(f_1)\,\text{size}(f_2)\,\text{size}(f_3)\\ &= O((n4^{\log n/\log 3}\log n)\exp(R\,dt^{1/d})(n/t)) = O(n^{R'})\end{aligned}$$

for some constant R'. □

THEOREM 5.4. *Let d be a fixed positive integer. For sufficiently large n and $1 \leqq t \leqq \min\{(\log n)^d, n-(\log n)^d\}$, there is a formula of depth $d+3$ that computes TH_t and the size of the formula is bounded by a polynomial in n.*

Proof. Let f be the formula as in Lemma 5.3. Let $\mathbf{G}$ equal x_i, where i is chosen uniformly at random between one and n inclusively. Let $A = TH_t^{-1}(0)$, and let $B = TH_t^{-1}(1)$. Then $\mathbf{G}$ is a $((t-1)/n, t/n)$ probabilistic separator for (A, B). Let $\mathbf{G}_1, \cdots, \mathbf{G}_s$, $s = \text{size}(f)$, be independent copies of $\mathbf{G}$. By Proposition 5.1, $f(\mathbf{G}_1, \cdots, \mathbf{G}_s)$ is a $(1/2^{n+2}, 1-1/2^{n+2})$ probabilistic separator for (A, B). By Proposition 5.2, there is a formula f' in the support of $f(\mathbf{G}_1, \cdots, \mathbf{G}_s)$ that is a $(0, 1)$ distributional separator for (A, B). That is, f' computes TH_t. By Lemma 5.3, we have that the size of f' is bounded by some polynomial in n. □

Appendix 1. For $2 < d \leqq \ln(m/c)$,

$$(d-1)(m/c)^{1/(d-1)} \geqq d(m/c)^{1/d}.$$

Proof. Since $d \leqq \ln(m/c)$, i.e., $e^d \leqq m/c$, and $(1+1/(d-1))^{d-1} \leqq e$,

$$\begin{aligned}(d-1)(m/c)^{1/(d-1)}/(d(m/c)^{1/d}) &= (1-1/d)(m/c)^{1/(d(d-1))}\\ &\geqq (1-1/d)\,e^{1/(d-1)}\\ &\geqq (1-1/d)(1+1/(d-1)) = 1.\end{aligned}$$

Thus, $(d-1)(m/c)^{1/(d-1)} \geqq d(m/c)^{1/d}$. □

Appendix 2. For $1/m < 1/c' < 1/c \leqq 1$ and $2 \leqq d$,

$$(d-1)(m/c')^{1/(d-1)} + (c'/c) \geqq d(m/c)^{1/d}.$$

Proof. Since $1/m < 1/c' < 1/c$, c' can be written as $(m/c)^h c = m^h c^{1-h}$ for some h with $0 < h < 1$, then

$$(d-1)(m/c')^{1/(d-1)} + c'/c = (d-1)(m/c)^{(1-h)/(d-1)} + (m/c)^h.$$

Let $G(h) = (d-1)(m/c)^{(1-h)/(d-1)} + (m/c)^h$. It is easy to see that $G(h)$ takes its minimum value when $G'(h) = 0$, which implies $h = 1/d$. Thus

$$\begin{aligned}(d-1)(m/c')^{1/(d-1)} + c'/c &\geqq G(1/d)\\ &= (d-1)(m/c)^{(1-1/d)/(d-1)} + (m/c)^{1/d}\\ &= (d-1)(m/c)^{1/d} + (m/c)^{1/d}\\ &= d(m/c)^{1/d}.\end{aligned}$$

□

Acknowledgments. We would like to thank the referees for careful comments and for pointing out additional references.

REFERENCES

[1] M. AJTAI AND M. BEN-OR, *A theorem on probabilistic constant depth computations*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1984, pp. 471–474.

[2] R. P. BOPPANA, *Amplification of probabilistic Boolean formulae*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 20-29.

[3] ———, *Threshold functions and bounded depth monotone circuits*, J. Comput. System Sci., 32 (1986), pp. 222-229.

[4] R. FAGIN, M. M. KLAWE, N. J. PIPPENGER, AND L. STOCKMEYER, *Bounded depth, polynomial-size circuits for symmetric functions*, Theoret. Comput. Sci., 36 (1985), pp. 239-250.

[5] L. DENENBERG, Y. GUREVICH, AND S. SHELAH, *Cardinalities definable by constant depth, polynomial size-circuits*, Tech. Report 26-83 (1983), Harvard University, Cambridge, MA, 1983.

[6] M. A. HARRISON, *Introduction to Switching and Automata Theory*, McGraw-Hill, New York, 1965, pp. 236-271.

[7] E. F. MOORE AND C. E. SHANNON, *Reliable circuits using less reliable relays*, J. Franklin Inst., 262 (1956), pp. 191-208 and pp. 281-297.

[8] L. G. VALIANT, *Short monotone formulae for the Majority function*, J. Algorithms, 5 (1984), pp. 363-366.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 56-87, February 1991
© 1991 Society for Industrial and Applied Mathematics
004

SIMPLE LOCAL SEARCH PROBLEMS THAT ARE HARD TO SOLVE*

ALEJANDRO A. SCHÄFFER† AND MIHALIS YANNAKAKIS‡

Abstract. Many algorithms for NP-hard optimization problems find solutions that are *locally optimal*, in the sense that the solutions cannot be improved by a polynomially computable perturbation. Very little is known about the complexity of finding locally optimal solutions, either by local search algorithms or using other indirect methods. Johnson, Papadimitriou, and Yannakakis [*J. Comput. System Sci.*, 37 (1988), pp. 79-100] studied this question by defining a complexity class PLS that captures local search problems. It was proved that finding a partition of a graph that is locally optimal into equal parts with respect to the acclaimed Kernighan-Lin algorithm is PLS-complete.

It is shown here that several natural, simple local search problems are PLS-complete, and thus just as hard. Two examples are: finding a partition that cannot be improved by a single swap of two vertices, and finding a stable configuration for an undirected connectionist network.

When edges or other objects are unweighted, then a local optimum can always be found in polynomial time. It is shown that the unweighted versions of the local search problems studied in this paper are P-complete.

Key words. local search, local optima, graph partitioning, connectionist networks, satisfiability, max cut, complexity theory, algorithms

AMS(MOS) subject classifications. 68Q15, 68Q20, 68Q25, 90C27

1. Introduction. One general approach to solving hard combinatorial optimization problems is to use a *local search* algorithm. The basic ingredient of a local search algorithm is a *neighborhood structure* that associates with every solution a set of "neighboring" solutions. Starting from some initial solution, the algorithm keeps moving to a better neighboring solution, until it arrives at a *locally optimal* solution, one that does not have a better neighbor. For example, a simple heuristic for the TRAVELING SALESMAN problem (TSP) is 2-OPT, a local search algorithm in which two tours are neighbors if one can be obtained from the other by exchanging one pair of edges for another pair. In the GRAPH PARTITIONING problem (partition the $2n$ vertices of a weighted graph into two equal-size sets to minimize the total weight of edges going from one set to the other), a simple neighborhood is the SWAP neighborhood, in which two partitions are neighbors if one can be obtained from the other by swapping two vertices.

In general, neighborhoods can be much more complicated. Two of the most sophisticated and successful local search algorithms are the Lin-Kernighan algorithm for TSP and the Kernighan-Lin algorithm for GRAPH PARTITIONING [16], [12]. The Lin-Kernighan heuristic generalizes the 2-OPT heuristic in that one "move" may consist of a sequence of exchanges of edges. In a similar way, in the Kernighan-Lin heuristic for GRAPH PARTITIONING, a move consists of a sequence of swaps of arbitrary length. In both cases, the heuristics use a nontrivial greedy criterion to prune the exponential-size search space of sequences of swaps or edge-exchanges. Any solution that is locally optimal with respect to the Kernighan-Lin criterion is certainly locally optimal with respect to the simpler SWAP criterion; the converse does not hold, which explains why the Kernighan-Lin heuristic produces better local optima.

* Received by the editors August 1, 1989; accepted for publication February 22, 1990.

† Department of Computer Science, Rice University, P.O. Box 1892, Houston, Texas 77251. This research was done while the author was visiting AT&T Bell Laboratories, Murray Hill, New Jersey.

‡ AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

There is empirical evidence that these (and other) local search algorithms converge quickly (e.g., [2], [8]), but very little has been proved about their running time, or, more generally, about the complexity of finding locally optimal solutions, possibly by other direct, noniterative methods. An interesting example in this regard is LINEAR PROGRAMMING, for which Simplex can be viewed as a local search algorithm, where local optimality coincides with global optimality. As in any local search problem, the running time of Simplex starting from a specific solution in a specific instance depends on the pivoting rule used, i.e., the rule for choosing a better neighboring (basic feasible) solution, if there is more than one. For many pivoting rules, there are bad examples that force Simplex to take exponential time; it is still an open problem whether there is a pivoting rule under which Simplex becomes a polynomial algorithm. However, in this case optimal solutions can be found in polynomial time by other direct methods such as the ellipsoid algorithm or Karmarkar's algorithm. (For a thorough introduction to LINEAR PROGRAMMING and an extensive list of references see [20].)

Johnson, Papadimitriou, and Yannakakis [9] raised the question of determining the complexity of finding locally optimal solutions. They defined a class called PLS (for Polynomial-time Local Search) that captures local optimization problems. We give a formal definition of PLS in the next section, but we note here that the most important condition for a local search problem to be in PLS is that one iteration of the local search algorithm takes polynomial time; i.e., we must be able to determine in polynomial time whether a solution is locally optimal and find a better neighbor if it is not. For example, the following problem is in PLS: "Given a TSP instance, find a solution that is locally optimal with respect to the 2-OPT heuristic." An algorithm for a PLS problem need not use the specified update heuristic to reach a locally optimal solution.

It was observed in [9] that the relationship of PLS to the more traditional classes P and NP is very unclear and probably difficult to resolve. On one hand, a problem in PLS cannot be NP-hard unless NP = co-NP, an event which is considered very unlikely. On the other hand, if all problems in PLS can be solved in polynomial time, then showing this would presumably require the discovery of a general-purpose algorithm for finding locally optimal solutions that should be at least as sophisticated as the ellipsoid algorithm or Karmarkar's algorithm. For this reason, Johnson, Papadimitriou, and Yannakakis proceeded by defining a notion of *reducibility* for PLS, and then by proving two important results: PLS has a generic complete problem (which we explain below), and finding a solution to GRAPH PARTITIONING that is locally optimal with respect to the celebrated Kernighan–Lin heuristic [12] is PLS-complete.

They suggested that local search problems with a simple neighborhood structure, such as 2-OPT for the TSP or SWAP for GRAPH PARTITIONING are most likely not PLS-complete, and gave the following reasoning. To decide if the "current solution" can be improved with a local perturbation, a local search algorithm must solve the subproblem of verifying local optimality. Johnson, Papadimitriou, and Yannakakis proved that this verification subproblem for the Kernighan–Lin algorithm is P-complete, and conjectured that this could in general be a necessary condition for PLS-completeness. Since the vericfiation subproblem for the SWAP heuristic is in LOGSPACE, they reasoned that GRAPH PARTITIONING with the SWAP criterion might not be PLS-complete.

Krentel [13] disproved their conjecture by exhibiting a PLS-complete problem whose verification subproblem is in LOGSPACE, and thereby refuted this line of reasoning. Specifically, Krentel showed that the problem of finding an assignment to a weighted SATISFIABILITY instance that cannot be improved by flipping a single variable is PLS-complete. For any assignment the number of neighboring assignments

is equal to the number of variables, and testing whether a flip improves the solution just requires summing the weights of satisfied clauses, so the verification problem for SATISFIABILITY can be solved in LOGSPACE.

We build on Krentel's result and prove that several natural, very simple local search problems are PLS-complete: We show that finding a locally optimal solution to GRAPH PARTITIONING under the SWAP heuristic is PLS-complete; this settles an open problem from [9] and [13] and provides an alternate proof for the PLS-completeness of the Kernighan–Lin heuristic. Another partitioning problem that is also PLS-complete is MAX CUT: Given a (weighted) graph, find a partition of its vertices, into two possibly unequal parts, so that the weight of the cut cannot be increased by moving a vertex from one side to the other. We show also that finding a locally optimal assignment for SATISFIABILITY with clauses of length at most 2 is PLS-complete.

A closely related problem is the problem of finding stable configurations in (discrete, undirected) connectionist networks, which as we show is also PLS-complete; this result addresses a problem of Haken, Luby, Godbeer, Lipscomb, and Parberry [5], [4], [17], [19]. A *connectionist* (or *neural*) *network* associates states with the vertices of a graph. A vertex may have to change its state depending on the value of a threshold function involving the states of neighboring vertices (we give formal definitions in the next section). A configuration is stable if no vertex needs to change its state. Such a model was introduced first by Hopfield in [6], where he showed that, under certain conditions, if the vertices asynchronously update their states, then they converge to a stable configuration. To prove convergence, he introduced a cost function and showed that a configuration is stable if and only if it is a local optimum for this cost function. In [6], Hopfield proposed the use of the model as a content addressable memory with error correction, where the stored words correspond to the stable configurations. Hopfield and Tank [7] proposed using a continuous, analogue version of the model for combinatorial optimization; Bruck and Goodman [1] suggested using the discrete version to find locally optimal solutions to problems such as MAX CUT.

Showing that a problem is PLS-complete means that it can simulate in principle any local search problem of PLS by choosing appropriate weights. It was observed in [9], and it is also true of the problems that we consider in this paper, that a further consequence of the completeness proof is that the standard local search algorithm for such a complete problem takes exponential time in the worst case.

Exponential running times are possible only when weights (of graph edges or clauses) are encoded in binary because this permits an exponential (in input size) range of feasible solution values. When weights are polynomially bounded, the range of solution values is polynomial and any reasonable local search algorithm will terminate in polynomial time. In this case, it is of interest to find alternative algorithms that may work fast in parallel. An interesting example is the MAXIMAL INDEPENDENT SET problem, which can be viewed as a local search problem, where we move from one independent set to another by adding a vertex. The standard local search algorithm adds one vertex at a time, and thus takes linear time. However, a maximal independent set can be constructed in NC by more sophisticated parallel methods [11], [18]. Luby [18] showed that the MAXIMAL INDEPENDENT SET problem is a special case of the stable configuration problem, if we choose appropriate small (polynomially bounded) weights in the connectionist network. Another special case, which generalizes MAXIMAL INDEPENDENT SET, is the DIFFERENT-THAN-MAJORITY LABELING problem (DTML): given a (unweighted) graph, find a partition of the vertices so that each vertex is on the opposite side of the majority of the vertices adjacent to it. The DTML problem is equivalent to the problem of finding a

locally optimal solution for MAX CUT on unweighted graphs. We prove that this problem is P-complete resolving a question of [18]. Similarly, we prove that the unweighted versions of GRAPH PARTITIONING under the SWAP neighborhood, and 2-SATISFIABILITY under the FLIP neighborhood are P-complete.

The rest of this paper is organized as follows. In § 2, we formally define PLS, describe the problems we consider, and summarize the relevant results from [9] and [13]. In § 3 we show simple reductions among the specific problems that we consider in this paper. Section 4 contains the P-completeness results for the unweighted problems, and § 5 contains the PLS-completeness proof for the weighted problems.

2. Definitions and previous results. The specification of a local search problem P includes a set of *instances* $\mathcal{I}$. For each instance $x \in \mathcal{I}$ we have a set of *feasible solutions* $\mathcal{F}(x)$, defined exactly as for NP. Each feasible solution $S \in \mathcal{F}(x)$ has an integer measure $\mu(S, x)$ that is to be maximized or minimized. In addition to its measure, every solution $S \in \mathcal{F}(x)$ has a set of *neighboring* solutions $\mathcal{N}(S, x)$. A solution S is *locally optimal* if it does not have a strictly better neighbor, i.e., one with a larger measure in case of a maximization problem, or smaller measure in case of a minimization problem. The local search problem is: given an input instance $x \in \mathcal{I}$, find a locally optimal solution.

A local search problem P is in the class PLS of *polynomial-time local search* problems if the following three polynomial-time algorithms exist.

(1) Algorithm A, on input $x \in \mathcal{I}$, computes an initial feasible solution belonging to $\mathcal{F}(x)$.

(2) Algorithm M, on input $x \in \mathcal{I}$ and $S \in \mathcal{F}(x)$, computes $\mu(S, x)$.

(3) Algorithm C, on input $x \in \mathcal{I}$ and $S \in \mathcal{F}(x)$, either determines that S is locally optimal or finds a better solution in $\mathcal{N}(S, x)$.

DEFINITION 2.1 [9]. A problem $P \in$ PLS is PLS-*reducible* to another problem $Q \in$ PLS if there are polynomial-time computable functions Φ and Ψ such that (a) Φ maps instances x of P to instances $\Phi(x)$ of Q, (b) Ψ maps (solution of $\Phi(x)$, x) pairs to solutions of x, and (c) for all instances x of P, if S is a locally optimal solution for the instance $\Phi(x)$ of Q, then $\Psi(S, x)$ is a locally optimal solution for the instance x of P.

PLS reductions usually have two desirable properties: first, they can be composed, and second, if we can find locally optimal solutions to Q in polynomial time, then we can also find locally optimal solutions for P in polynomial time. In all our PLS reductions, the functions Φ and Ψ are not only polynomial-time computable, but also in LOGSPACE.

We now formally introduce the array of search problems we consider. In all problems, except the one on connectionist networks, weights and measures are nonnegative. We define each problem by specifying its set of instances, the feasible solutions, the measure function (as with an optimization problem), and the neighborhood structure. The existence of the polynomial-time algorithms A, M, C required for membership in PLS is obvious in all the cases.

DEFINITION 2.2. An instance of FLIP is a Boolean circuit with n input bits and n output bits. A feasible solution is an assignment of $\{0, 1\}$ to the input bits. Given such an assignment, the circuit computes values for the output bits. The measure of a solution is the sequence of output bits treated as a binary number. Two solutions are neighbors if one can be obtained from the other by flipping the value of a single input bit. There are two versions of the problem, depending on whether the measure is to be maximized or minimized; the two versions are equivalent.

DEFINITION 2.3. An instance of WEIGHTED CNF SATISFIABILITY, or SAT for short, is a Boolean formula in CNF with a positive integer weight on each clause.

A feasible solution is an assignment (of 0 or 1) to all the variables. The measure of a feasible solution is the sum of the weights of the satisfied clauses. The neighborhood of a feasible solution contains all solutions obtained by flipping the value of one variable. This is a maximization problem.

DEFINITION 2.4. An instance of WEIGHTED kCNF SATISFIABILITY, or kSAT for short, is an instance of SAT in which every disjunctive clause contains at most k literals. Solutions, measures, and neighborhoods are defined as for SAT.

DEFINITION 2.5. An instance of WEIGHTED NOT ALL EQUAL kSATISFIABILITY, or NAE kSAT for short, consists of clauses with at most k literals of the form NAE $(x_1, \cdots, x_k)$, where each x_i is a literal or a constant (0 or 1). Such a clause is satisfied if its constituents do not all have the the same value. Each clause is assigned a positive integer weight. Measures and neighborhoods are defined as for SAT. The restriction of NAE kSAT to those instances that do not contain any negative literals is called POS NAE kSAT.

DEFINITION 2.6. An instance of the local search problem for Graph Partitioning under the SWAP neighborhood consists of a simple undirected graph $G = (V, E)$, with an even number $2n$ of vertices and weights on the edges. A feasible solution is a partition of V into two sets V_1, V_2 of equal size. The measure of a solution is the weight of the cut, i.e., the total weight of the edges having one endpoint in each half of the partition. It is traditional to define an optimal solution as a solution of *minimum* weight; as noted in [9], the minimization and maximization versions of the problem are equivalent, both with respect to finding global and local optima. In the SWAP neighborhood, two solutions are neighbors if one can be obtained from the other by swapping one element of V_1 with one element of V_2.

We can define a more sophisticated neighborhood for the Graph Partitioning problem, namely, the neighborhood that is explored by the Kernighan-Lin heuristic [12]. In this heuristic, we move from a partition to a neighboring partition by a sequence of (at most n) greedy swaps. In each step of the sequence, we examine all pairs of vertices from opposite sides that have not changed side since the beginning of the sequence, and choose to swap the best such pair: "best" in the sense that the weight of the cut decreases the most or increases the least. The neighbors of the original partition are the partitions obtained after the first, second, $\cdots$, nth swap of this sequence. (This description covers the basic structure of the Kernighan-Lin heuristic; a fuller specification must include also a rule for breaking possible ties among best swaps in each step, and also a rule for moving to a better neighboring partition if there are two or more such neighbors.) If a partition is not locally optimal with respect to the SWAP neighborhood, that is, if it has an improving swap, then the Kernighan-Lin heuristic will discover this and perform such a swap in the first step of the sequence. Thus the Kernighan-Lin algorithm explores a (much) larger neighborhood. Note that, in general, finding a locally optimal solution with respect to a smaller neighborhood structure is at least as easy as finding a solution with respect to the larger one (though the quality of the solution may not be as good). Hence PLS-completeness for the smaller neighborhood structure implies completeness for the larger one. We let KL-GRAPH PARTITIONING be the problem of finding a solution that is locally optimal with respect to the Kernighan-Lin algorithm.

Fiduccia and Mattheyses [3] proposed a variant of the Kernighan-Lin heuristic for which the running time of one iteration is significantly smaller. Dunlop and Kernighan [2] implemented both heuristics in a practical setting; they reported that the Fiduccia-Mattheyses heuristic generally reaches a local optimum faster than the Kernighan-Lin heuristic, but sometimes finds inferior local optima. The Fiduccia-Mattheyses heuristic also uses a sequence of steps to move from one partition to a

neighboring one, but the steps are somewhat different, in that vertices are moved from one side to the other one at a time. This heuristic motivates the following local search problems.

DEFINITION 2.7. Instances, feasible solutions, and measures of FM-GRAPH PARTITIONING are defined as for the Graph Partitioning problem. Note that the measure (weight of a cut) can be computed even for partitions into unequal size sets although these are infeasible. The neighborhood of a solution contains all solutions reachable by a sequence of (at most n) steps, where each step consists of the following two substeps: in the first substep, we examine all the vertices that have not moved since the beginning of the sequence, and choose to move the best such vertex from one side to the other; in the second substep, we move from the opposite side the best as-yet-unmoved vertex, so that the partition becomes again balanced. In both substeps, "best" means again that the move decreases the most (or increases the least) the weight of the cut. We can define also another much simpler local search problem, which we call FM-SWAP, with a smaller neighborhood: each partition has just one neighbor, the partition obtained after the first step of the Fiduccia–Mattheyses heuristic. Thus FM-SWAP bears the same relationship to FM-GRAPH PARTITIONING as SWAP to KL-GRAPH PARTITIONING.

The idea of considering temporary intermediate moves to infeasible solutions is also used in the Lin–Kernighan heuristic for TSP [16]. Strictly speaking, the definition of FM-GRAPH PARTITIONING should include a rule for breaking ties in each step among intermediate solutions of equal weight, so that we do not consider an exponential size set of equal-weight intermediate solutions. Also, in reality we may allow the partition to become more unbalanced temporarily. However, we show that finding a local optimum for the simpler FM-SWAP neighborhood is PLS-complete, and therefore, the same is true of any reasonable variant of FM-GRAPH PARTITIONING regardless of the tie-breaking rule.

DEFINITION 2.8. An instance of MAX CUT consists of a simple undirected graph $G=(V, E)$ with positive weights on the edges. A feasible solution is a partition of V into two sets V_1, V_2 (not necessarily of equal size). Measure is defined as for GRAPH PARTITIONING, but optimal solutions have *maximum* measure. Two solutions are neighbors if one can be obtained from the other by moving a single vertex from one side of the partition to the other side.

DEFINITION 2.9. An instance of (CONNECTIONIST MODEL) STABLE CONFIGURATION is an undirected graph with a weight w_e on each edge and a threshold t_v for each vertex. Weights and thresholds may be negative in this problem. A *configuration* is an assignment to each vertex v of a *state* $s_v \in \{-1, 1\}$. The state of a vertex v stable in a configuration if $s_v = 1$ and $\sum_{(u,v)\in E} w_{u,v}s_u + t_v \geqq 0$ or $s_v = -1$ and $\sum_{(u,v)\in E} w_{u,v}s_u + t_v \leqq 0$. A configuration is *stable* if the states of all the vertices are stable. Stable configurations coincide with the locally maximal solutions with respect to the following measure:

$$\sum_{(u,v)\in E} w_{u,v}s_u s_v + \sum_{v\in V} s_v t_v.$$

Two configurations are neighbors if they differ in the state of exactly one vertex. (Some authors use 0 and 1 for the states, and/or require the minimization rather than maximization of the measure; all these versions are equivalent [4], [17].)

In the course of proving their main results, Johnson, Papadimitriou, and Yannakakis also considered the complexity of three types of problems associated with PLS problems. The first is the *verification problem*, mentioned in § 1, which is the decision problem solved by algorithm C: Is a feasible solution locally optimal? The

second is the *running time problem*: What is the worst-case time complexity (as a function of instance size over all instances and starting solutions) of the standard local search algorithm that finds a locally optimal solution by repeatedly applying algorithm C. The third problem is the *standard algorithm problem*: Given an instance and an initial solution S, find the local optimum that would be produced by the standard algorithm starting from S. Of course, the answers to the second and third problems depend on the particular choices made by algorithm C in each iteration, in case more than one neighbor offer an improvement. However, for the results of [9] as well as our results, it does not matter how C chooses among better neighbors; i.e., the results apply regardless of the choices of C.

We now formally summarize the previous results.

THEOREM 2.10 [9]. *If a* PLS *problem is* NP-*hard, then* NP = co-NP.

THEOREM 2.11 [9]. FLIP *is* PLS-*complete. The verification problem for* FLIP *is* P-*complete, and the standard algorithm problem is* NP-*hard. The standard algorithm for* FLIP *takes exponential time in the worst case.*

THEOREM 2.12 [9]. KL-GRAPH PARTITIONING *is* PLS-*complete. The verification problem for* KL-GRAPH PARTITIONING *is* P-*complete, and the standard algorithm problem is* NP-*hard. The standard algorithm for* KL-GRAPH PARTITIONING *takes exponential time in the worst case.*

THEOREM 2.13 [13]. SAT *is* PLS-*complete.*

It follows from the proof of Theorem 2.13 that the standard algorithm for SAT takes also exponential time in some cases, and the standard algorithm problem is NP-hard. However, the verification problem for SAT is in LOGSPACE [13], and hence very unlikely to be P-complete.

3. Some simple reductions. In this section, we present some simple reductions among the problems defined in § 2. All the reductions are both PLS and LOGSPACE reductions. Besides PLS-completeness, we also want to prove that there are instances where the standard algorithm takes exponential time to reach a local optimum. Johnson, Papadimitriou, and Yannakakis proved such a result for FLIP and KL-GRAPH PARTITIONING in an ad hoc fashion. We shall give a sufficient condition for reductions to preserve the exponential computation time property. We need first some definitions.

DEFINITION 3.1. Let P be a PLS problem and let I be an instance of P. The *neighborhood graph* NG (I) of the instance I is a directed graph with one vertex for each feasible solution to I, and with an arc $s \to t$ whenever $t \in \mathcal{N}(s, I)$. The *transition graph* TG (I) is the subgraph that includes only those arcs $s \to t$ for which $\mu(t, I)$ is strictly better than $\mu(s, I)$ (i.e., greater if P is a maximization problem, and smaller if P is a minimization problem). The *height* of a vertex v is the length of the shortest path in TG (I) from v to a sink (a vertex with no outgoing arcs). The *height* of TG (I) is the largest height of a vertex.

We will be concerned only with the transition graph. This graph depends only on the measure and the neighborhood structure; it does not depend on the particular algorithms A, M, C required in the definition of PLS. First, note that the transition graph is acyclic; the measure induces a topological ordering of the vertices (the arcs are directed from worse to better vertices). The local optima are the sinks of the graph. The transition graph TG (I) represents the possible legal moves for algorithm C on instance I. Starting from some initial vertex v, the standard local search algorithm follows some path to a sink. Thus the height of v is a lower bound on the number of iterations needed by the standard algorithm. If there are instances whose transition graphs have exponential height, then the standard algorithm takes exponential time

in the worst case, regardless of how it chooses better neighbors. This is actually the case for FLIP and KL-GRAPH PARTITIONING [9], and we show a similar result for our problems. The NP-hardness of the standard algorithm problem follows from that of the following problem: "Given an instance I and a solution s, compute a locally optimal solution (i.e., a sink of TG (I)) that is reachable from s." We refer to *this* problem as the "standard algorithm problem" from now on.

DEFINITION 3.2. Let P, Q be PLS problems, and let (Φ, Ψ) be a PLS reduction from P to Q. We say that the reduction is *tight* if for any instance I of P we can choose a subset $\mathcal{R}$ of feasible solutions for the image instance $J = \Phi(I)$ of Q so that the following properties are satisfied:

(1) $\mathcal{R}$ contains all local optima of J.

(2) For every feasible solution p of I, we can construct in polynomial time a solution $q \in \mathcal{R}$ of J such that $\Psi(q, I) = p$.

(3) Suppose that the transition graph of J, TG (J) contains a directed path $q \to\to q'$, such that $q, q' \in \mathcal{R}$, but all internal path vertices are outside $\mathcal{R}$, and let $p = \Psi(q, I)$ and $p' = \Psi(q', I)$ be the corresponding feasible solutions of I. Then, either $p = p'$ or TG (I) contains an arc from p to p'.

It is easy to see that tight reductions compose. We usually refer to the solutions in $\mathcal{R}$ as the *reasonable* solutions of the instance J.

LEMMA 3.3. *Suppose that P and Q are problems in* PLS, *and that Φ, Ψ define a tight* PLS-*reduction from problem P to problem Q*:

(1) *If I is an instance of P and $J = \Phi(I)$ is the image instance of Q, then the height of* TG (J) *is at least as large as the height of* TG (I). *Thus if the standard algorithm of P takes exponential time in the worst case, then so does the standard algorithm for Q.*

(2) *If the standard algorithm problem for P is* NP-*hard, then the same is true for Q.*

Proof. (1) Let I be an instance of P, let TG (I) be its transition graph, and let p be a feasible solution (vertex) whose height is equal to the height of TG (I). Let $J = \Phi(I)$, and let $q \in \mathcal{R}$ be a feasible solution of J such that $\Psi(I, q) = p$. We claim that the height of q in TG (J) is at least as large as the height of p in TG (I). To see this, consider a shortest path from q to a sink of TG (J), and let the vertices of $\mathcal{R}$ that appear on this path be $q, q_1, \cdots, q_k$. Let $p_1, \cdots, p_k$ be the images under Ψ of these solutions, i.e., $p_i = \Psi(q_i, I)$. From the definition of a tight reduction, we know that q_k is a local optimum of J, and thus p_k is a local optimum of I. Also, for each i, either $p_i = p_{i+1}$ or there is an arc in TG (I) from p_i to p_{i+1}. Therefore there is a path of length at most k from vertex p to a sink of TG (I).

(2) Given an instance I of P and a solution p, we can find a locally optimal solution (sink) that is reachable from p as follows. First, construct the instance J and the solution $q \in \mathcal{R}$ as above. Then, solve the standard algorithm problem for Q to obtain a locally optimal solution q_k of J that is reachable from q, and finally return the solution $p_k = \Psi(q_k, I)$. As we argued above, p_k is a local optimum of I that is reachable from p. □

We proceed now to show some simple reductions.

LEMMA 3.4. *The problems* MAX CUT *and* POS NAE 3SAT *can be reduced to each other via tight* PLS *reductions.*

Proof. We reduce first MAX CUT to POS NAE 3SAT. Let the (weighted) graph G be an instance of MAX CUT. We construct an instance I of POS NAE 3SAT that has one variable for every vertex of G and one clause NAE (u, v) for every edge (u, v) of G with the same weight. A truth assignment for the variables of I induces in a natural way a partition of G, namely, the partition that contains on one side the vertices that correspond to true variables and on the other side the vertices that correspond to false variables. The weight of the cut of this partition is equal to the weight of the

clauses of I that are satisfied by the truth assignment. It is straightforward to verify that this is a tight PLS reduction if we let $\mathcal{R}$ be the set of all truth assignments.

We reduce now in the opposite direction. Let I be an instance of POS NAE 3SAT. We construct a graph G having one vertex for each variable of I and in addition two more vertices labeled with the constants 0 and 1. We have an edge $(0, 1)$ with "huge" weight $3L$, where L is the total weight of all the clauses of the instance I. In addition, for every clause of length two, NAE (x, y) with weight W in I, where x and y are variables or constants, we include an edge (x, y) in G with the same weight W. For every length three clause NAE (x, y, z) with weight W, we include three edges (x, y), (y, z), (x, z), with weight $W/2$ each. If a pair of vertices appears in more than one clause, then the weight of the edge joining them is the sum of the weights arising from all the clauses that contain both of them. Let us call a partition of G *reasonable* if the vertices 0 and 1 are on opposite sides, and let $\mathcal{R}$ be the set of reasonable partitions. First observe that if a partition is not reasonable, then it can be improved by moving vertex 0 (or vertex 1) to the other side to gain the very heavy edge $(0, 1)$ that dominates all other edges. A reasonable partition induces a truth assignment for the variables of I, where a variable is 1 (true) if its vertex is on the same side as vertex 1, and 0 (false) if it is on the same side as vertex 0. A clause of length 2 is satisfied if and only if the corresponding edge is in the cut. Furthermore, if a clause of length 3 is satisfied then two of the corresponding three edges are in the cut, and if it is not satisfied then it does not contribute any edges to the cut. Thus the weight of a reasonable partition is equal to the weight of the corresponding assignment for I plus $3L$ (the weight of the edge $(0, 1)$). Furthermore, moving a variable vertex from one side to the other in a reasonable partition corresponds to flipping the variable in the truth assignment. It follows that a locally optimal partition of G induces a locally optimal assignment for I. We can verify that $\mathcal{R}$ satisfies the conditions of Definition 3.2. □

LEMMA 3.5. *There are tight* PLS *reductions from* MAX CUT *to the following problems*: (a) 2SAT, (b) SWAP, (c) FM-SWAP, (d) STABLE CONFIGURATION.

Proof. Let the weighted graph G be an instance of MAX CUT.

(a) We construct an instance I of 2SAT that has one variable for every vertex of G. If G has an edge (x, y) with weight W, then we include in I clauses $(x \vee y)$ and $(\bar{x} \vee \bar{y})$, each of weight W. Note that if in a truth assignment we have $x = y$, then only one of these two clauses is satisfied, whereas if $x \neq y$ then both clauses are satisfied. A truth assignment of I induces a partition of G that contains on one side the vertices corresponding to true variables and on the other side the vertices corresponding to false variables. The weight of the satisfied clauses is equal to the weight of the cut plus the total weight of all the edges. Thus when we flip a variable in the 2SAT instance, I changes the weight of the assignment by the same amount as the weight of the cut is changed by moving the corresponding vertex to the other size. Therefore a locally optimal truth assignment induces a locally optimal partition. Choosing $\mathcal{R}$ to be the set of all assignments satisfies the tightness conditions.

(b) We reduce first to the maximization version of SWAP, and then reduce the maximization to the minimization version. Let H be the graph that consists of two copies G_1, G_2 of G with the same weights on the edges, and that also includes edges with "huge" weight L connecting every vertex x_1 of G_1 with its corresponding vertex x_2 of G_2. By "huge" we mean that the weight L of these edges exceeds the total weight of the edges of G. We say that a partition of H into equal parts is *reasonable* if every pair x_1, x_2 of corresponding vertices is separated, and we let $\mathcal{R}$ be the set of reasonable partitions. If a partition of H into equal parts is not reasonable, then there are vertices u and v of G such that one side of the partition contains both copies u_1, u_2 of u, and the other side contains both copies v_1, v_2 of v. In this case, swapping u_1 and v_1 increases

the weight of the cut. Thus we may restrict attention to reasonable partitions of H. Such a partition induces in a natural way a partition of G by restricting it to the vertices of G_1 (equivalently, G_2). If G has n vertices, then the weight of the partition of H is equal to $n \cdot L$ plus twice the weight of the induced partition of G. The only swaps that can possibly improve a reasonable partition of H are swaps of corresponding vertices; swapping such a pair x_1, x_2 of vertices of H corresponds to moving the vertex x to the other side in the induced partition of G. It follows that a locally optimal partition into equal parts of H under the SWAP neighborhood induces a locally optimal partition of G for MAX CUT. Also, our choice of $\mathcal{R}$ satisfies the tightness conditions.

We now reduce the maximization version of SWAP to the minimization version. Regard H as being the complete graph where the missing edges have weight 0, and let L be the maximum edge weight. Let H' be the graph on the same vertices where the weight of every edge is equal to L minus its weight in H; edges with 0 weight in H' can be regarded as not being present. Observe that if we partition the complete graph on $2n$ vertices into two equal halves, there are always n^2 edges in the cut. Thus any partition into two equal halves that weighs U in H must weigh $n^2L - U$ in H'. A swap that gains weight T for the maximization version loses weight T for the minimization version. Thus the set of locally optimal partitions and the computation paths are exactly the same in both versions.

(c) As we will see, the same reduction of SWAP works also for FM-SWAP. First, we reduce to the maximization version by constructing the same graph H. If a partition is not reasonable, then the best vertex to move in the first substep is a vertex u_1 that is on the same side as its corresponding vertex u_2, in order to gain the very heavy edge (u_1, u_2). In the second substep, the best vertex that can be moved is a vertex v_1 from the new side of u_1 which is on the same side as the corresponding vertex v_2. Thus a locally optimal partition of H must be reasonable. Suppose now that we have a reasonable partition of H and consider its neighboring partition. If the first move involves a copy of vertex x of G, then we lose an edge of weight L, and therefore the second move involves the other copy of x, i.e., the neighboring partition is obtained by swapping the two copies of x. If moving vertex x in the induced partition of G changes the weight of the cut by T, then moving in H a copy of x changes the weight of the cut by $T - L$. Since we choose in the first substep of FM-SWAP to move the best vertex, we conclude that if $T > 0$ for some vertex x of G, then we will swap in H two copies of such a vertex for an overall increase of $2T$ in the cut of H. Thus a locally optimal partition of H with respect to the FM-SWAP neighborhood induces a locally optimal partition of G for MAX CUT. Again, our choice of $\mathcal{R}$ satisfies the tightness conditions.

We reduce the maximization to the minimization version of FM-SWAP as above by constructing the same graph H'. Note that if we have a partition of the complete graph into two equal halves and we move a vertex from one side to the other then the number of edges in the cut changes from n^2 to $(n+1)(n-1)$, regardless of the chosen vertex. If moving a vertex increases the weight by T in the maximization version, then moving the same vertex decreases the weight by $T + L$ in the minimization version. Thus the best vertex to move first is the same in both versions. The same holds for the second move and thus locally optimal partitions with respect to the two versions are identical.

(d) The instance of STABLE CONFIGURATION consists of the same graph G, where all the vertices have threshold 0, and the weight of every edge is the negative of its weight in G. Let W be the total weight of the edges of G. A configuration for the connectionist network induces a partition of G, where the one side contains the

vertices in state +1 and the other side the vertices in state −1. Flipping the state of a vertex in a configuration corresponds to moving the vertex to the other side in the induced partition of G. An easy calculation shows that the weight of a configuration is equal to $2T - W$, where T is the weight of the cut of the induced partition of G. It follows that a stable configuration induces a locally optimal partition of G. Choosing $\mathcal{R}$ to be the set of all configurations satisfies the tightness conditions. □

LEMMA 3.6. *The unweighted* MAX CUT *problem can be* PLS-*reduced to the unweighted version of* 2SAT, SWAP, FM-SWAP, *and to* STABLE CONFIGURATION *with all edge weights* −1.

Proof. In the case of 2SAT and STABLE CONFIGURATION, the reductions of the previous lemma suffice. For SWAP and FM-SWAP we need different reductions. We reduce first to the maximization versions. Let the unweighted graph G be an instance of MAX CUT with n vertices. Construct a graph H that consists of G and $3n$ more isolated vertices. Consider a partition of H into two equal parts and the induced partition of G. Note that the partition of H contains isolated vertices on both sides. If the partition of G is not locally optimal but can be improved by moving a vertex x to the other side, then we can improve the partition of H by swapping x with an isolated vertex from the other side; such a swap will be performed both in the case of the SWAP and the FM-SWAP neighborhood. Thus a locally optimal partition of H into equal parts under either of the two neighborhoods induces a locally optimal partition of G for MAX CUT. (Note. This PLS reduction is not tight.)

To reduce the maximization version to the minimization version we just construct the complement H' of H. The arguments are similar to those of the previous lemma. □

We note that finding a stable configuration for connectionist networks with edge weights +1, or more generally, with positive weights, is an easier problem. The special case of STABLE CONFIGURATION where all the edge weights are positive is equivalent to finding a local optimum for a MIN CUT problem [4], [17]. Thus in the positive weight case, even a global optimum can be constructed in polynomial time, even in the case of arbitrary weights. Furthermore, in the case of polynomially bounded weights, the MIN CUT problem can be solved in Random NC [10].

4. P-completeness results. We prove in this section that finding a locally optimal solution to the unweighted MAX CUT problem is P-complete, and then deduce the same result for the unweighted versions of the other problems defined in § 2.

The reduction is from the generic P-complete problem CIRCUIT VALUE [15]. An instance of CIRCUIT VALUE consists of a Boolean circuit C and an assignment of values to the inputs of C; the problem is to compute the output of the circuit on the given input assignement. We may assume without loss of generality that all the gates of C are NOR gates with fanin 2. We number the gates in topological order, but for technical reasons, we use only even indices. That is, the gates are numbered $g_2, g_4, \cdots, g_{2n}$ so that for each gate g_i, its two inputs, denoted $I_1(g_i)$ and $I_2(g_i)$ are either inputs of the circuit or are the outputs of lower-numbered gates. The correct value of each gate g_i is $g_i = \neg(I_1(g_i) \vee I_2(g_i))$.

We give the reduction in two stages. First we reduce to a restricted case of weighted POS NAE 3SAT with polynomially bounded integer weights, and then we reduce to MAX CUT as in Lemma 3.4 and show how to remove the weights altogether. The restricted case of POS NAE 3SAT is characterized by the following property. There is a weight associated with every variable of the constructed instance I, so that the instance of MAX CUT constructed from I as in Lemma 3.4 satisfies the following two

conditions: (1) the weight of an edge connecting two variable vertices is the product of the weights of the variables, and (2) the weight of an edge connecting a variable vertex to the vertex 0 or 1 is a multiple of the weight of the variable.

Our instance I of POS NAE 3SAT includes (among others) variables $g_2, g_4, \cdots, g_{2n}$ corresponding to the gates of the circuit. It has the property that in any locally optimal truth assignment all gate variables g_i have the correct value, consistent with the circuit C and the given input assignment. If we were allowed to use exponentially large weights, then this property could be achieved rather easily without using any other variables by including for each gate g_i a set of clauses stating that g_i has the correct value, with the weights of the clauses decreasing exponentially with the index i [9]. Johnson, Papadimitriou, and Yannakakis [9] used the fact that the NOR-gate constraint $g_i = \neg(I_1(g_i) \vee I_2(g_i))$ can be expressed by the following three NAE 3SAT clauses:

$$\mathrm{NAE}\,(I_1(g_i), g_i, 1) \wedge \mathrm{NAE}\,(I_2(g_i), g_i, 1) \wedge \mathrm{NAE}\,(I_1(g_i), I_2(g_i), g_i).$$

These clauses are simultaneously satisfied if and only if g_i has the correct value. The first two clauses say that if an input is 1 then g_i must be 0, and the third clause says that if both inputs are 0 then g_i is 1. If a gate variable g_i is incorrect, then we can gain at least one of its clauses by correcting it; in doing so, we may lose some clauses from higher gates that have g_i as their input, but because of the exponential weights the overall effect of the flip is positive.

The exponential weights in the above reduction "mask" the fanout of the gates; i.e., a gate can be corrected without worrying about higher gates that it feeds. We cannot use exponential weights in this reduction, but we instead use extra variables to achieve the same masking effect. The most important variables, which we call the *control* variables, are z_i and y_i, for $1 \leqq i \leqq 2n$. Typically $z_i = \neg y_i$, but we need both variables to make all variable occurrences positive. In a locally optimal assignment, all z_i will be 0 and all y_i will be 1; we refer to these as the *natural* values for these variables. Besides the control variables, we have for every even index i (i.e., for each gate g_i) a number of *local* variables. Local variables participate in clauses with some z and y variables, but they do not participate in clauses with variables for gates other than g_i and the inputs to g_i. Local variables are denoted by Greek letters.

Before describing the clauses in detail, we give a brief overview of the properties of the instance I. If an assignment is not consistent with the circuit, say gate variable g_i is the lowest incorrect gate, then the standard local search algorithm will fix the assignment as follows. First, it will move to an assignment where for $j \leqq i-1$, the control variables have their natural value $z_j = 0$, $y_j = 1$ (and the corresponding gates are correct), and for $j \geqq i+1$ the control variables have their unnatural value $z_j = 1$, $y_j = 0$. Second, the local variables for gates higher than $i+1$ will assume certain "default" values such that flipping an input to a higher gate g_j (such as the incorrect gate variable g_i) will have no effect on the clauses for g_j. Third, the local variables for the gate g_i will assume certain other values that encourage correcting the value of the variable g_i. Fourth, the standard algorithm will correct the value of g_i, reset z_i and y_i to their natural values, and proceed similarly to fix the variables with higher indices in order, one index at a time.

We describe now the instance I in detail. We shall denote the weight of a variable v by $|v|$. For each $i = 1, \cdots, 2n$, we let $|z_i| = 100(2n+1-i)$, $|y_i| = 100(2n+1-i)+50$, and for even i, $|g_i| = 100(2n+1-i)+60$. Furthermore, for every even index i, we have variables α_i^1, α_i^2, δ_i^1, δ_i^2 of weight $|g_i|+10 = 100(2n+1-i)+70$, variables β_i^1, β_i^2, β_i^3,

γ_i^1, γ_i^2, γ_i^3 of weight equal to $|g_i|$, and a variable ω_i of weight $|\delta_i^1| - |g_i| = 10$. There are a few more variables that we define later.

We partition the clauses into five types, A through E, that serve conceptually different purposes. For every gate g_i we have a set of type A clauses involving the above local variables. The purpose of these clauses is to test if g_i has an incorrect value, and if this is the case, to set z_i or y_i to its unnatural value. The type A clauses are shown graphically in Fig. 4.1. There is one NAE 3SAT clause for every triangle of the figure. We list the clauses and specify their weights below. The vertices are arranged in each part of the figure from top to bottom according to weight. As we shall see, parts (a) and (b) cause z_i to be set to 1 if an input is 1 and g_i is also 1, and part (c) causes y_i to be 0 if both inputs are 0 but g_i is also 0.

First, we have clauses

$$\text{NAE}\,(I_1(g_i), \alpha_i^1, 1) \wedge \text{NAE}\,(I_1(g_i), \delta_i^1, 0)$$

of weight $2|I_1(g_i)| \cdot |\alpha_i^1| = 2|I_1(g_i)| \cdot |\delta_i^1|$; if $I_1(g_i)$ is an input to the circuit, then we substitute in the clause the corresponding constant of the input assignment, and we let its weight in the above expression be $100(2n+1)+60$ (i.e., the weight of g_0 if there were such a gate). The multiplier 2 is needed because when we transform to MAX CUT, we replace a 3-literal clause by three edges whose weight is $\frac{1}{2}$ the weight of the clause (see the proof of Lemma 3.4). We have the analogous clauses for the second input:

$$\text{NAE}\,(I_2(g_i), \alpha_i^2, 1) \wedge \text{NAE}\,(I_2(g_i), \delta_i^2, 0)$$

of weight $2|I_2(g_i)| \cdot |\alpha_i^2| = 2|I_2(g_i)| \cdot |\delta_i^2|$. Next, we have clauses

$$\text{NAE}\,(\alpha_i^1, \beta_i^1, 0) \wedge \text{NAE}\,(\alpha_i^2, \beta_i^2, 0) \wedge \text{NAE}\,(\delta_i^1, \delta_i^2, \beta_i^3),$$

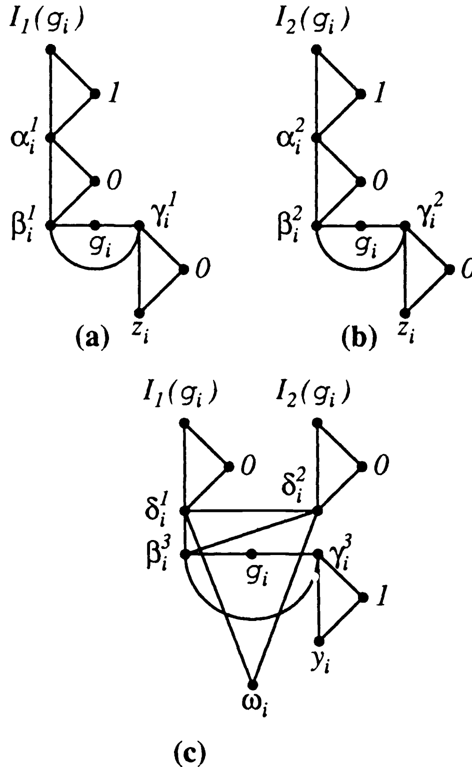


FIG. 4.1. *The clauses of type* A; *each triangle represents one clause.*

all of the same weight $2|\alpha_i^1| \cdot |\beta_i^1|$, and also a clause

$$\text{NAE}(\delta_i^1, \delta_i^2, \omega_i)$$

of weight $2|\delta_i^1| \cdot |\omega_i|$. Next, we have clauses

$$\text{NAE}(\beta_i^1, g_i, \gamma_i^1) \wedge \text{NAE}(\beta_i^2, g_i, \gamma_i^2) \wedge \text{NAE}(\beta_i^3, g_i, \gamma_i^3)$$

of weight $2|g_i|^2$. Finally, we have clauses $\text{NAE}(\gamma_i^1, z_i, 0)$, $\text{NAE}(\gamma_i^2, z_i, 0)$ of weight $2|g_i| \cdot |z_i|$, and a clause $\text{NAE}(\gamma_i^3, y_i, 1)$ of weight $2|g_i| \cdot |y_i|$.

The purpose of the type B clauses is to propagate an unnatural value for a control variable (0 for y_i, 1 for z_i) to the control variables with higher index. We depict graphically the type B clauses in Fig. 4.2; the vertical position of the vertices indicates their weight, and there is one clause for every triangle. For each $i = 1, \cdots, 2n$ we have a clause $\text{NAE}(y_i, z_i, 0)$ of weight $2|y_i| \cdot |z_i|$, and for each $i = 1, \cdots, 2n-1$ we have a clause $\text{NAE}(z_i, y_{i+1}, 1)$ of weight $2|z_i| \cdot |y_{i+1}|$.

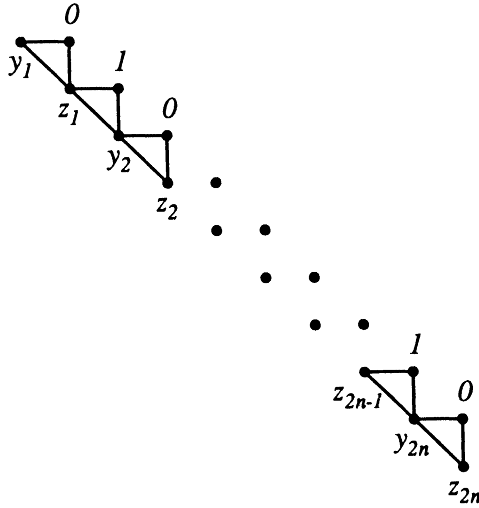


FIG. 4.2. *The clauses of type* B; *each triangle represents one clause.*

The type C clauses encourage setting the control variables to their natural values. For each $i = 1, \cdots, 2n$ we have a clause $(y_i \neq 0)$ of weight $40|y_i|$, and a clause $(z_i \neq 1)$ of weight $40|z_i|$.

The type D clauses encourage appropriate default values for the local variables corresponding to each gate g_i. These values depend on the value of z_{i-1} and y_{i-1}. We partition the local variables into two groups. The Y group includes α_i^1, α_i^2, γ_i^1, γ_i^2, β_i^3, ω_i; in case of ties from the clauses of higher type, we would like these variables to be equal to y_{i-1}. The Z group includes the rest of the local variables: β_i^1, β_i^2, γ_i^3, δ_i^1, δ_i^2; we would like these variables to be equal to z_{i-1}. We can express equality between two variables in POS NAE 3SAT by saying that they are both different from a third variable. Specifically, we introduce six variables $\psi_i^1, \cdots, \psi_i^6$ corresponding to the six variables of the Y group, and another five variables $\zeta_i^1, \cdots, \zeta_i^5$ corresponding to the five variables of the Z group; all these variables have weight 3. Every variable u of the Y or Z group is connected with an inequality clause of weight $3|u|$ to its corresponding ψ or ζ variable. In addition, for each variable ψ_i^j we have a clause $(y_{i-1} \neq \psi_i^j)$ of weight $3|y_{i-1}|$, and for each variable ζ_i^j we have a clause $(z_{i-1} \neq \zeta_i^j)$ of weight $3|z_{i-1}|$.

Finally, the type E clauses encourage correcting the gate variables. As we shall show later, if z_{i-1} and y_{i-1} have their natural values, then the variables α_i^1 and α_i^2 are equal to the negations of the inputs $I_1(g_i)$ and $I_2(g_i)$, respectively. For each even index i, we introduce a variable ρ_i of weight 1, and clauses $(\alpha_i^1 \neq \rho_i)$, $(\alpha_i^2 \neq \rho_i)$, $(\rho_i \neq 0)$ of weight $|\alpha_i^1| = |\alpha_i^2|$, and a clause $(\rho_i \neq g_i)$ of weight $|g_i|$. The inequality clauses of ρ_i are depicted in Fig. 4.3. This concludes the specification of the NAE SAT instance I.

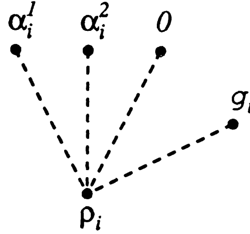


FIG. 4.3. *The clauses of type* C; *each dashed line represents one clause.*

We now prove four lemmas that characterize the local optima of our POS NAE 3SAT instance. Each lemma has an implicit dependent clause: "In any locally optimal assignment." Some of the lemma proofs are subdivided into simpler claims. The basic method of each proof is that a solution not having the claimed properties can be improved by some flip. In each lemma proof we indicate which clauses are used; that is, which clauses can be gained by a flip if the properties claimed in the lemma are not satisfied. In the next section, we will reuse the same construction, except that we will omit the type C clauses, and we will increase the weight of types A and B clauses by multiplying the uniformly by a large constant. As we shall see, the first three lemmas here do not use the type C clauses, although we have to make sure that they do not get in the way, i.e., that they do not prevent a flip from being profitable. Also, the lemmas hold if we increase proportionately the weights of types A and B clauses. In the PLS reduction of the following section we will reuse the first three lemmas verbatim.

LEMMA 4.1. *If gate g_i is incorrect, then $z_i = 1$. If $y_i = 0$, then $z_i = 1$. If $z_i = 1$, then for all $j > i$, $y_j = 0$, $z_j = 1$.*

Proof. The proof uses the clauses of type A and B.

CLAIM 4.1A. *If $I_1(g_i) = 1$, then $\alpha_i^1 = 0$ and $\beta_i^1 = 1$. If $I_2(g_i) = 1$, then $\alpha_i^2 = 0$ and $\beta_i^2 = 1$.*

Proof. The clause NAE $(I_1(g_i), \alpha_i^1, 1)$ weights more than all the other clauses of α_i^1 combined. Thus if $I_1(g_i) = 1$, then we can flip, if necessary, α_i^1 to 0 to satisfy this clause. The clause NAE $(\alpha_i^1, \beta_i^1, 0)$ weighs more than all the other clauses of β_i^1 combined. Since $\alpha_i^1 = 0$, we can flip β_i^1 to 1 to satisfy this clause. Similarly, we can argue that if $I_2(g_i) = 1$, then $\alpha_i^2 = 0$ and $\beta_i^2 = 1$. □

CLAIM 4.1B. *If $I_1(g_i) = 1$, but $g_i = 1$, then $\gamma_i^1 = 0$. If $I_2(g_i) = 1$, but $g_i = 1$, then $\gamma_i^2 = 0$.*

Proof. The clause NAE $(\beta_i^1, g_i, \gamma_i^1)$ weighs more than all the other clauses of γ_i^1 combined. If $I_1(g_i) = 1$, then $\beta_i^1 = 1$ by the previous claim. Thus if $g_i = 1$, then we will flip γ_i^1 to 0 to gain the clause NAE $(\beta_i^1, g_i, \gamma_i^1)$. The proof for γ_i^2 is analogous. □

CLAIM 4.1C. *If $\gamma_i^1 = 0$ or $\gamma_i^2 = 0$, then $z_i = 1$.*

Proof. We gain one of the clauses

$$\text{NAE}\,(\gamma_i^1, z_i, 0) \wedge \text{NAE}\,(\gamma_i^2, z_i, 0)$$

by flipping z_i to 1. In doing so, we may lose the type B clause NAE $(z_i, y_{i+1}, 1)$, the

type C clause ($z_i \neq 1$), and type D clauses; the combined weight of these clauses is smaller than the weight of the type A clause we gain. □

CLAIM 4.1D. *If* $I_1(g_i)=0$, *then* $\delta_i^1=1$. *If* $I_2(g_i)=0$, *then* $\delta_i^2=1$.

Proof. The clause NAE $(I_1(g_i), \delta_i^1, 0)$ weighs more than all the other clauses of δ_i^1 combined. If $I_1(g_i)=0$, then we gain this clause by flipping δ_i^1 to 1. The proof for δ_i^2 is analogous. □

CLAIM 4.1E. *If both inputs to* g_i *are* 0, *then* $\beta_i^3=0$ *and* $\omega_i=0$.

Proof. Suppose $I_1(g_i)=I_2(g_i)=0$. By Claim 4.1D, $\delta_i^1=\delta_i^2=1$. We can gain the clauses NAE $(\delta_i^1, \delta_i^2, \beta_i^3)$ and NAE $(\delta_i^1, \delta_i^2, \omega_i)$ by flipping β_i^3 and ω_i to 0. Any clauses lost are significantly lighter. □

CLAIM 4.1F. *If* $g_i=\beta_i^3=0$. *then* $\gamma_i^3=1$.

Proof. We can gain the clause NAE $(g_i, \beta_i^3, \gamma_i^3)$ by flipping γ_i^3 to 1. This clause weighs more than all the other clauses of γ_i^3 combined. □

CLAIM 4.1G. *If* $I_1(g_i)=I_2(g_i)=g_i=0$, *then* $y_i=0$.

Proof. By Claim 4.1E, $\beta_i^3=0$. By Claim 4.1F, $\gamma_i^3=1$. We can gain the clause NAE $(y_i, \gamma_i^3, 1)$ by flipping y_i to 0. In doing so, we may lose the clause NAE $(y_i, z_i, 0)$ of type B, the clause $(y_i \neq 0)$ of type C, and clauses of type D. The combined weight of these clauses is smaller than the weight of the type A clause we gain. □

CLAIM 4.1H. *If* $y_i=0$, *then* $z_i=1$. *If* $z_i=1$, *then for any* $j>i$, $y_j=0$ *and* $z_j=1$.

Proof. First observe that flipping a variable z_i to 1, and a variable y_j to 0 does not lose any type A clauses. If $y_i=0$ and $z_i=0$, then by flipping z_i to 1 we gain the type B clause NAE $(y_i, z_i, 0)$ of weight $2|y_i| \cdot |z_i|$, possibly lose the next type B clause NAE $(z_i, y_{i+1}, 1)$ of weight $2|z_i| \cdot |y_{i+1}|$, and we lose also the type C clause $(z_i \neq 1)$ of weight $40|z_i|$ and six type D clauses that weigh collectively less than $20|z_i|$. Thus the total loss is no more than $(2|y_{i+1}|+60)|z_i|$, which is less than the gain of $2|y_i| \cdot |z_i|$. By a similar argument, if $z_i=1$ and $y_{i+1}=1$, then flipping y_{i+1} to 0 results in a net gain. □

CLAIM 4.1I. *If gate* g_i *is incorrect, then* $z_i=1$.

Proof. If the gate has a 1 input and a 1 output, then the claim follows from Claims 4.1B and 4.1C. If the gate has two 0 inputs and a 0 output, then the claim follows from Claims 4.1G and 4.1H. □

The lemma follows from Claims 4.1H, and 4.1I. □

LEMMA 4.2. *Suppose that* $y_{i-1}=0$ *and* $z_{i-1}=1$ *for some even index i. Then flipping either input of the gate* g_i *does not affect the type* A *clauses corresponding to* g_i.

Proof. From the previous lemma we know that $z_i=1$ and $y_i=0$. Each of the inputs $I_1(g_i)$, $I_2(g_i)$ of the gate g_i occurs in two clauses (both of type A) of g_i:

$$\text{NAE}\,(I_1(g_i), \alpha_i^1, 1) \wedge \text{NAE}\,(I_1(g_i), \delta_i^1, 0),$$

$$\text{NAE}\,(I_2(g_i), \alpha_i^2, 1) \wedge \text{NAE}\,(I_2(g_i), \delta_i^2, 0).$$

We prove that $\alpha_i^1=\alpha_i^2=0$ and $\delta_i^1=\delta_i^2=1$, which together imply the lemma. The proof uses types A and D clauses.

CLAIM 4.2A. *The* ψ_i^j *variables are* 1, *and the* ζ_i^j *variables are* 0 *for all j.*

Proof. Every ψ_i^j variable is connected by inequality clauses to y_{i-1} and to a local variable of gate g_i. Since the clause with y_{i-1} weighs more because y_{i-1} has larger weight than the local variables of g_i, all the ψ_i^j variables will have value opposite to that of y_{i-1}. Similarly, the ζ_i^j will have value opposite to that of z_{i-1}. □

Thus the type D clauses "push" or "bias" towards 0 the local variables of the Y group $(\alpha_i^1, \alpha_i^2, \gamma_i^1, \gamma_i^2, \beta_i^3, \omega_i)$ and push towards 1 the local variables of the Z group $(\delta_i^1, \delta_i^2, \beta_i^1, \beta_i^2, \gamma_i^3)$. We show that, in fact, the local variables of g_i have the values towards which they are biased by type D.

CLAIM 4.2B. $\alpha_i^1=0$, $\beta_i^1=1$, *and* $\gamma_i^1=0$. *Similarly,* $\alpha_i^2=0$, $\beta_i^2=1$, *and* $\gamma_i^2=0$.

Proof. From the type D clauses, the variables β_i^1 and γ_i^1 are biased towards opposite values: the first towards 1 and the second towards 0. These two variables belong to a common clause NAE $(\beta_i^1, g_i, \gamma_i^1)$, and in addition each is contained in one more clause that would not be violated by the preferred value. The additional clause of β_i^1 contains a 0, and that of γ_i^1 contains z_i which has value 1. If $g_i = 1$, then we can set γ_i^1 to its preferred value 0, after which we can set β_i^1 to its preferred value 1. Similarly, if $g_i = 0$, we can first set $\beta_i^1 = 1$, and then $\gamma_i^1 = 0$. Thus regardless of the value of g_i, we have $\beta_i^1 = 1$ and $\gamma_i^1 = 0$. Since both type A clauses of α_i^1 contain a 1, we can set it to its preferred value 0. The argument for the variables with superscript 2 is analogous. □

CLAIM 4.2C. *$\delta_i^1 = \delta_i^2 = 1$, $\beta_i^3 = \omega_i = 0$, and $\gamma_i^3 = 1$.*

Proof. Suppose first that both δ_i^1 and δ_i^2 are 0. Then we can flip one of them to its preferred value 1. We do not lose any type A clauses, and we gain a type D clause. Thus we may assume that at least one of the δ variables is 1. It follows then that ω_i will be set to its preferred value 0 because flipping ω_i to 0 does not lose the type A clause NAE $(\delta_i^1, \delta_i^2, \omega_i)$.

Consider now the variables β_i^3 and γ_i^3. They have opposite preferred values (0 for the first, 1 for the second), belong to the common clause NAE $(\beta_i^3, g_i, \gamma_i^3)$, and in addition, each belongs to one more type A clause that contains a value opposite to the preferred value: NAE $(\delta_i^1, \delta_i^2, \beta_i^3)$ contains a 1, and NAE $(\gamma_i^3, y_i, 1)$ contains y_i which is 0. As in the previous claim, we can argue that regardless of g_i we must have $\beta_i^3 = 0$ and $\gamma_i^3 = 1$. Since all the clauses of the variables δ_i^1, δ_i^2 contain a 0, we can conclude now that *both* these variables take on their preferred value 1. □

The next lemma is a counterpart to Lemma 4.2 when $y_{i-1} = 1$ and $z_{i-1} = 0$.

LEMMA 4.3. *Suppose $y_{i-1} = 1$ and $z_{i-1} = 0$ for some even index i. If the gate variable g_i is correct, then flipping z_i and y_i does not affect the type A clauses. If g_i is incorrect, then flipping g_i does not affect the type A clauses corresponding to the gate g_i, and furthermore, gains a type E clause.*

Proof. The variable g_i occurs as a gate output in three type A clauses NAE $(\beta_i^j, g_i, \gamma_i^j)$, $j = 1, 2, 3$, corresponding to the gate g_i, and to one type E clause $(g_i \neq \rho_i)$. We show that $\rho_i = I_1(g_i) \vee I_2(g_i)$, which implies that flipping g_i to the correct value gains a type E clause. We also show that if g_i is incorrect, then for each $j = 1$, 2, 3, we have $\gamma_i^j = \neg\beta_i^j$, which makes it possible to flip g_i without losing the type A clauses in which it is an output. The variable z_i belongs to two type A clauses NAE $(z_i, \gamma_i^1, 0)$, NAE $(z_i, \gamma_i^2, 0)$, and y_i belongs to one type A clause NAE $(y_i, \gamma_i^3, 1)$. We show that if g_i is correct, then $\gamma_i^1 = \gamma_i^2 = 1$ and $\gamma_i^3 = 0$, which makes it possible to flip z_i and y_i without losing their type A clauses.

The proof uses types A, D, and E clauses. As in the previous lemma, we can argue that the ψ variables have value 0 (opposite to y_{i-1}), and the ζ variables have value 1 (opposite to z_{i-1}). Thus the local variables are now *biased* by type D in the opposite direction: variables α_i^1, α_i^2, γ_i^1, γ_i^2, β_i^3, ω_i prefer 1, and β_i^1, β_i^2, γ_i^3, δ_i^1, δ_i^2 prefer 0. However, the *actual* values of the local variables may now disagree with their biases and may instead depend on the inputs of the gate, since the local variables and the inputs occur together in type A clauses.

CLAIM 4.3A. *$\alpha_i^1 = \neg I_1(g_i)$ and $\beta_i^1 = I_1(g_i)$. Similarly, $\alpha_i^2 = \neg I_2(g_i)$ and $\beta_i^2 = I_2(g_i)$. Finally, $\rho_i = I_1(g_i) \vee I_2(g_i)$.*

Proof. Suppose first that $I_1(g_i) = 1$. Then, as in Lemma 4.1, we will set α_i^1 to 0 to satisfy the clause NAE $(I_1(g_i), \alpha_i^1, 1)$, and then we will set β_i^1 to 1 to satisfy the clause NAE $(\alpha_i^1, \beta_i^1, 0)$. Suppose now that $I_1(g_i) = 0$. Then both type A clauses of α_i^1 contain a 0, and thus α_i^1 will be set to 1, its preferred value. Consider the variables β_i^1 and γ_i^1:

they have opposite preferred values (0 for the first, 1 for the second), they belong to the common clause NAE $(\beta_i^1, g_i, \gamma_i^1)$, and in addition, each belongs to one more type A clause that contains a value opposite to its preferred value. By an argument similar to one used in the proof of Claim 4.2B, we can conclude that, regardless of g_i, we have $\beta_i^1 = 0$ and $\gamma_i^1 = 1$.

Thus for any value of $I_1(g_i)$, we have $\alpha_i^1 = \neg I_1(g_i)$ and $\beta_i^1 = I_1(g_i)$. By a symmmetric argument, $\alpha_i^2 = \neg I_2(g_i)$ and $\beta_i^2 = I_2(g_i)$. Consider the variable ρ_i now. It occurs in four inequality clauses of type E:

$$(\alpha_i^1 \neq \rho_i) \wedge (\alpha_i^2 \neq \rho_i) \wedge (\rho_i \neq 0) \wedge (\rho_i \neq g_i).$$

If both inputs to g_i are 0, then we just proved that both α_i^1 and α_i^2 are 1; as a consequence, ρ_i will be set to 0 regardless of g_i because the α_i's have larger weight than g_i. If at least one input is 1, then the corresponding α_i variable is 0, and ρ_i will be set to 1. □

CLAIM 4.3B. $\beta_i^3 = I_1(g_i) \vee I_2(g_i)$.

Proof. As in Lemma 4.1, if an input is 0, then the corresponding δ_i^j variable will be set to 1 to satisfy the clause NAE $(I_j(g_i), \delta_i^j, 0)$. Thus if both inputs are 0, then both δ_i variables are 1, and the variables β_i^3, ω_i are set to 0 to satisfy the clauses NAE $(\delta_i^1, \delta_i^2, \beta_i^3)$ and NAE $(\delta_i^1, \delta_i^2, \omega_i)$.

Suppose now that at least one input, say $I_1(g_i)$, is 1. First, we claim that at least one of the δ_i variables is 0, for if both are 1, then we can flip δ_i^1 to its preferred value 0 to gain a type D clause without losing any clauses. Consider now the variables β_i^3 and γ_i^3. The only clauses heavier than type D containing them are the type A clauses:

$$\text{NAE}\,(\delta_i^1, \delta_i^2, \beta_i^3) \wedge \text{NAE}\,(\gamma_i^3, g_i, \beta_i^3) \wedge \text{NAE}\,(\gamma_i^3, y_i, 1).$$

We may set β_i^3 to its preferred value 1 without losing the first clause because one of the δ_i variables is 0. The variable γ_i^3 can be set to 0 (its preferred value) without losing the third clause because that clause has a 1. By an argument similar to the one in the proof of Claim 4.2B we can set $\beta_i^3 = 1$ and $\gamma_i^3 = 0$ in an order that does not lose the second clause. □

CLAIM 4.3C. *If g_i is correct, then $\gamma_i^1 = \gamma_i^2 = 1$ and $\gamma_i^3 = 0$. If g_i is incorrect, then for each $j = 1, 2, 3$ we have $\gamma_i^j = \neg\beta_i^j$.*

Proof. Suppose first that g_i is correct. By Claim 4.3A, $\beta_i^1 = I_1(g_i)$, and therefore at least one of the variables β_i^1 and g_i must be 0. This implies that both type A clauses of γ_i^1 have a 0, and therefore, we can flip γ_i^1 to its preferred value 1. Similarly, we can conclude that $\gamma_i^2 = 1$. Furthermore, since $\beta_i^3 = I_1(g_i) \vee I_2(g_i)$, by Claim 4.3B, and g_i is correct, i.e., $g_i = \neg\beta_i^3$, we can set γ_i^3 to its preferred value 0 without losing the type A clause NAE $(\beta_i^3, g_i, \gamma_i^3)$.

Suppose now that g_i is incorrect. If for some j we have $\beta_i^j = \gamma_i^j$, then it must be the case that $g_i = \neg\beta_i^j$, because otherwise we would flip γ_i^j to gain the clause NAE $(\beta_i^j, g_i, \gamma_i^j)$. If $j = 3$, this would imply that g_i is correct, because β_i^3 is the OR of the inputs. Suppose that $j = 1$; the argument for $j = 2$ is symmetric. Since $\beta_i^1 \neq g_i$ and γ_i^1 prefers the value 1, we conclude that $\gamma_i^1 = \beta_i^1 = 1$, and $g_i = 0$, that is, g_i has the correct value. □

The lemma follows by combining all the claims above with the first paragraph of the proof. □

We will now use the clauses of type C to finish the proof that POS NAE 3SAT is P-complete.

LEMMA 4.4. *Every gate is correct, every z is 0, and every y is 1.*

Proof. By Lemma 4.1, if the conclusion of the lemma does not hold, then $z_i = 1$ for some i. Let i be the smallest such index. We shall derive a contradiction by showing that $z_i = 0$.

Suppose first that i is even, i.e., there is a gate g_i. By Lemma 4.1 and our choice of i the variables z_{i-1} and y_{i-1} exist ($i \geqq 2$), and have their natural values 0 and 1, respectively. Suppose that the gate variable g_i is incorrect. By Lemma 4.3, we can gain a type E clause by flipping g_i to its correct value, without losing any clauses corresponding to g_i. The variable g_i may occur also in clauses corresponding to higher gates g_j, namely, those gates that have gate g_i as an input. Since we use only even indices for the gates, such higher gates g_j have $j \geqq i+2$, and therefore $y_{j-1} = 0$ and $z_{j-1} = 1$. By Lemma 4.2, flipping the variable g_i does not lose any clauses corresponding to higher gates that may have g_i as their input. We conclude that the gate variable g_i is correct. By Lemma 4.3, we do not lose any type A clauses by flipping z_i and y_i to their natural value.

Suppose now that i is odd or that i is even and g_i is correct. Assume that $y_i = 0$. By flipping y_i to 1, we gain a type C clause. We do not lose a type B clause because $z_{i-1} = 0$ We may only lose type D clauses, but these weigh much less. We conclude therefore that $y_i = 1$. It follows now that flipping z_i to 0 is profitable for a similar reason: we gain a type C clause, we do not lose a type B clause because $y_i = 1$, and may only lose lighter type D clauses. □

Thus we have shown that in a locally optimal truth assignment of the constructed instance I, all gate variables have the correct value. It remains to reduce the instance I of POS NAE 3SAT to an instance of unweighted MAX CUT. From I we can construct a weighted graph H as in the proof of Lemma 3.4. Recall that H has one vertex for every variable, and two vertices labeled 0 and 1. The weights of the clauses of the instance I were chosen carefully so that the weighted graph H satisfies the conditions that we mentioned earlier: (1) an edge connecting two variable vertices u and v has weight equal to the product $|u| \cdot |v|$ of the weights of the variables, and (2) the weight of the edge (if it exists) connecting a variable vertex v to a constant vertex 0 or 1 is a multiple of the weight $|v|$ of v. The graph H has the property that any locally optimal partition induces a locally optimal truth assignment for I.

Let v be a variable vertex of H. From properties (1) and (2), the total weight of its incident edges is a multiple of $|v|$, say $d(v) \cdot |v|$. We call $d(v)$ the *degree* of v. We argue first that we may restrict ourselves to the case where all variable vertices of H have odd degree. For suppose this is not the case. Form a new instance of MAX CUT by multiplying all the vertex weights by 2, all the edge weights by 4, and adding an edge from each variable vertex v to, say, vertex 0 of weight equal to the new weight of v (i.e., twice the old weight). If the old degree of v was $d(v)$, the new degree is $2d(v)+1$. Any locally optimal partition in the new graph is also locally optimal in the old graph: if moving a variable vertex v from one side to the other changes the cut in the old graph by $c|v|$ (it must be a multiple of $|v|$), then it changes the cut in the new graph by at least $(2c-1)2|v|$. If the change is positive in the old graph ($c > 0$), then it is also in the new one.

We assume now that all vertices of H have odd degree, and complete the transformation to the unweighted MAX CUT problem. Let M be the total weight of the edges incident to the constant vertices 0 and 1. We construct a graph G as follows. We have a set V_0 and a set V_1 of $4M+1$ vertices each corresponding to the constants 0 and 1, respectively. These vertices induce a complete bipartite graph $V_0 \times V_1$. We replace every variable vertex v of H by a set N_v of $|v|$ vertices. An edge (u, v) connecting two variable vertices is replaced by a complete bipartite graph between N_u and N_v.

If a variable vertex v is connected in H to the constant vertex 0 or 1 by an edge of weight $c|v|$, then we connect every vertex of N_v to c vertices of V_0 or V_1.

Consider a locally optimal partition of the graph G. At least $3M$ vertices from V_0 are connected only to V_1; since these vertices of V_0 have the same neighbors and have odd degree, they must be on the same side of the partition, namely, the side opposite to the majority of V_1. It follows then that all vertices of V_1 must be on the opposite side from the $3M$ vertices of V_0, which implies further that, in fact, all the vertices of V_0 are also together. Consider now the vertex set N_v that replaced a variable vertex v of H. Except for edges to vertices of V_0 and V_1, all the vertices of N_v have the same neighbors; since there is an odd number of neighbors, we conclude that all vertices of N_v are on the same side. Thus a locally optimal partition of G induces a partition of H. We claim that the partition of H is also locally optimal. For, suppose that moving the variable vertex v of H changes the cut by $c|v|$; then moving in the partition of G any vertex of N_v to the other side will change the cut by c. Since the partition of G is locally optimal, $c \leqq 0$. Thus we have proved Theorem 4.5.

THEOREM 4.5. *Finding a locally optimal partition for unweighted* MAX CUT *is* P-*complete.* □

Combining this result with the reductions in § 3 we get Corollary 4.6.

COROLLARY 4.6. *Finding a local optimum for the unweighted versions of the following problems is* P-*complete*: 2SAT, SWAP, FM-SWAP, FM-GRAPH PARTITIONING, NAE 3SAT; *also,* STABLE CONFIGURATION *with edge weights* -1 *is* P-*complete.* □

5. PLS-completeness results. In this section we give a PLS reduction from FLIP to POS NAE 3SAT, showing the second problem is PLS-complete. Then we use the reductions of § 3 to show the same result for the other problems.

Let the Boolean circuit C be an instance of FLIP. Let the input variables of C be $v_1, \cdots, v_p$, and let the outputs be $c_0, \cdots, c_m$; that is, the weight of an input assignment is $\Sigma_j 2^j c_j$. We construct an instance I of POS NAE 3SAT that includes (among others) variables $v_1, \cdots, v_p$, such that the values of these variables in a locally optimal assignment for the instance I form a locally optimal solution to the FLIP instance C.

As in the previous section, we assume without loss of generality that all gates of C are NOR gates with fanin 2. We again number the gates with even indices in topological order as $g_2, g_4, \cdots, g_{2n}$. We assume also that the circuit computes the negations of the outputs as well, and we denote them as $\hat{c}_0, \cdots, \hat{c}_m$. Both the c_i and the $\hat{c}_i$ gates are included among the g's.

Besides the given circuit C, we use p additional circuits $T_1, \cdots, T_p$, called *test circuits*, one for each input variable v_i. The idea for these test circuits comes from [13]. They are used to precompute the consequences of a variable flip. For each $i = 1, \cdots, p$, the circuit T_i is identical to C, and has the same input variables except for the variable v_i in whose place it has a different variable w_i. The instance I that we construct includes clauses that encourage w_i to be the negation of v_i. In this case, T_i computes what the output of the circuit C would be, were we to flip the variable v_i. We use two subscripts to denote the gates of the test circuits; the copy of gate g_j in T_i is denoted $g_{i,j}$. The copies of the outputs c_j and their negations $\hat{c}_j$ in T_i are denoted by $t_{i,j}$ and $\hat{t}_{i,j}$, respectively.

The construction of the POS NAE 3SAT instance I is quite complicated. Before plunging into its details, we describe some of the more important new variables. First, the reduction includes many copies of the POS NAE 3SAT construction of the previous

section: for each circuit (the circuit C and the test circuits T_i), we include one variable corresponding to every gate. In addition, we have control variables z and y as in the previous section, as well as the local variables denoted by Greek letters. Furthermore, for each input variable v_i, we introduce variables d_i and e_i. The variable d_i is intended to indicate whether we have decided to flip v_i to improve the solution of the FLIP instance. The variable d_i is set to 1 when we first decide to flip v_i, but d_i gets reset to 0 before all the consequences of the flip are taken care of. The variable e_i is intended to indicate whether $v_i = w_i$; i.e., $e_i = 1$ if $v_i = w_i$. In a locally optimal solution, all d_i and e_i variables will be 0, and all gate variables will be correct (i.e., consistent with the circuits and the input assignment).

Suppose that we have an assignment in which all $e_i = d_i = 0$, $v_i \neq w_i$, and all gates are correct, but the input assignment is not locally optimal for the FLIP instance, say flipping v_i is advantageous. Then the variables associated with the flip will change value in the following order. First d_i is set to 1. Second, the control variables y and z for the circuit C and the test circuits T_r with $r \neq i$ (i.e., the circuits that have v_i as input) are set to their unnatural values. Third, v_i is flipped. Fourth, e_i is set to 1. Fifth, all the gates in the circuits C and T_r for $r \neq i$ are corrected, and the control variables are reset to their natural values. Sixth, d_i is reset to 0. Seventh, the control variables for the circuit T_i are set to their unnatural value. Eighth, w_i is flipped back to being the opposite of the new v_i value. Ninth, e_i is reset to 0. Tenth, all the gates of T_i are corrected, and the control variables are reset, finishing the cycle.

Our instance I will have NAE clauses of lengths 2 and 3. For clarity, we write the clauses of length 2 as inequalities. We have eleven types of clauses, numbered from 1 to 11. The weight of the clauses decreases exponentially with the type. Thus, for example, one clause of type 1 is worth more than all the clauses of types 2–11 combined.

Our construction involves some more variables besides the ones we mentioned. Because of the positivity restriction we cannot use negations in the clauses. Therefore we introduce auxiliary variables to play the role of negations; for example, we have variables $\hat{d}_i$ and $\hat{e}_i$ playing the roles of $\neg d_i$ and $\neg e_i$. Furthermore, because the clauses must be in NAE form and not in the usual SAT form, and because they must have length at most 3, we have to use some gadgets of extra variables and clauses to achieve succinctness. Such extra variables are of a local nature, i.e., each occurs in only one clause type, and are denoted by Greek letters. We now define the instance I of POS NAE 3SAT. We assume that n is sufficiently large ($n > 15$ will do), so that a clause of one type weighs more than all the clauses of strictly lower types combined.

The heaviest clauses are of type 1. They ensure that at most one d variable can be 1. For each i, such that $1 \leqq i \leqq p$ and each $j \neq i$, there is a clause NAE $(d_i, d_j, 1)$ of weight 2^{37n}.

The clauses of type 2 concern the circuit C. They test if the gate variables of C are correct, and if not they turn some d_i to 1, and eventually help to fix the incorrect gates at the appropriate time. The clauses of type 2 include types A, B, D, and E clauses of the P-completeness reduction from the previous section. The type C clauses, which are *not* included here, played a role in setting the control variables to their natural values. We have other clauses below that, depending on the values of the d and e variables, flip the control variables at the proper times.

Corresponding to types A, B, D, and E from the last section, we have here identical clauses of types 2A, 2B, 2D, and 2E. The only difference is that previously, if an input to a gate g was an input to the circuit, then we substituted in the type A clauses the corresponding constant of the given input assignment for the CIRCUIT VALUE problem; here, we are not given such an assignment and we instead substitute the

corresponding input variable v_i. The clauses of types 2A and 2B have the same relative weights as in the last section, but we multiply the weights by the same constant so that they fall in the range between 2^{36n} and 2^{33n}. The clauses of types 2D and 2E have the same relative weights as in the last section, but we *do not* multiply their weights; these clauses have very small weights, less than 2^n, which is smaller than the weight of even the type 11 clauses.

Besides the clauses from the previous section, type 2 includes some additional clauses, which we call type 2C. Recall that the last gate of the circuit C has index $2n$. We have one more y variable, y_{2n+1}, and we have the following clauses of type 2C: NAE $(z_{2n}, y_{2n+1}, 1)$ of weight 2^{31n}, a clause NAE $(y_{2n+1}, d_i, 0)$ of weight 2^{30n} for each $i = 1, \cdots, p$, and a clause $(y_{2n+1} \neq 1)$ of weight $(p-1)2^{30n}$. The purpose of these clauses is as follows. Recall from the previous section that, if a gate is incorrect, then all subsequent control variables are set to their unnatural value; thus z_{2n} is set to 1 and y_{2n+1} to 0. The fundamental goal of the type 2C clauses is to set some d_i to 1 in this case. Because of the type 1 clauses, only one d_i can be set profitably to 1; therefore if $y_{2n+1} = 0$, then exactly $p-1$ of the clauses NAE $(y_{2n+1}, d_i, 0)$ will be violated. The purpose of the last clause $(y_{2n+1} \neq 1)$ is to compensate for the $p-1$ lost clauses.

The clauses of type 3 play a similar role for the test circuits T_i. That is, for each $i = 1, \cdots, p$, we have types 3A, 3B, 3D, and 3E clauses for the circuit T_i. The type 3A and 3B clauses are the same as in the P-completeness reduction, except that their weight is multiplied by an appropriate constant so that their weight falls between 2^{29n} and 2^{25n}. Again, if an input to a gate is a circuit input, then in the corresponding type A clauses we use the variable v_j or w_i as appropriate. The weight of types 3D and 3E clauses is not multiplied; thus these clauses have small weight, less than 2^n. We use two subscripts to denote the variables of the type 3 clauses: the first subscript i indicates the circuit T_i to which the variable refers, and the second subscript is as before. For example, $z_{i,k}$ denotes the kth control variable z of the circuit T_i.

In addition to the above clauses, we have for each $i = 1, \cdots, p$ one clause of type 3C: NAE $(z_{i,2n}, d_i, 1)$ of weight 2^{24n}, where $z_{i,2n}$ is the z variable corresponding to the last gate of T_i. The fundamental goal of these clauses is to ensure that $d_i = 0$, if T_i has an incorrect gate.

The clauses of type 4 measure the effect of flipping a variable v_i. When $d_i = 1$, then the output is picked from the circuit T_i, but if all d_i are 0, then the output is picked from the circuit C. For each pair i, j with $i = 1, \cdots, p$ and $j = 0, 1, \cdots, m$, we have the following type 4 clauses:

$$\text{NAE}\,(d_i, c_j, 0), \qquad \text{NAE}\,(d_i, \hat{t}_{i,j}, 1)$$

of weight $2^{20n} \cdot 2^j$. When $d_i = 1$, we may want to flip c_j to implement the gate corrections resulting from a variable flip. However, this could affect the $p-1$ clauses NAE $(d_r, c_j, 0)$ for $r \neq i$. To cancel any positive or negative effect of this change we have a clause $(c_j \neq 1)$ that weighs $p-1$ times as much as the corresponding length 3 clauses; that is, it weighs $(p-1) \cdot 2^{20n} \cdot 2^j$.

The clauses of type 5 force the variable $\hat{d}_i$ to act as the negation of d_i. We have a clause NAE $(d_i, \hat{d}_i, 1)$ of weight 2^{19n}, and a clause $(\hat{d}_i \neq 0)$ of weight 2^{18n}.

The clauses of type 6 give credit for performing an anticipated flip. If $d_i = 1$, we would like to flip the value of v_i. That is, we would like to (momentarily) force $v_i = w_i$. If we were reducing to 3SAT, then we could just write the implication $d_i \Rightarrow (v_i = w_i)$ as a 3CNF formula. Expressing the implication in POS NAE 3SAT requires a more complicated gadget, which we depict graphically in Fig. 5.1. Each solid triangle represents a 3-variable clause, while each dashed edge represents a 2-variable clause.

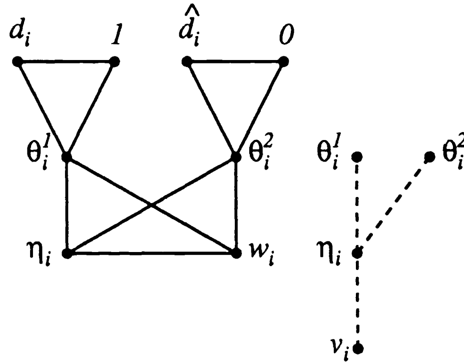


FIG. 5.1. *Gadget for type 6; triangles and dashed edges represent clauses.*

We have the following clauses for each $i=1, \cdots, p$. First,

$$\text{NAE}(d_i, \theta_i^1, 1) \wedge \text{NAE}(\hat{d}_i, \theta_i^2, 0)$$

of weight 2^{17n} each,

$$\text{NAE}(\theta_i^1, \eta_i, w_i) \wedge \text{NAE}(\theta_i^2, \eta_i, w_i)$$

of weight 2^{16n} each, and

$$(\eta_i \neq v_i)$$

of weight 2^{15n}. Finally,

$$(\theta_i^1 \neq \eta_i) \wedge (\theta_i^2 \neq \eta_i)$$

of weight 2^{14n} each.

The clauses of type 7 give credit for setting e_i to 1, when $v_i = w_i$ and credit for resetting e_i to 0 otherwise. In this case also we need a gadget in order to use POS NAE 3SAT form; see Fig. 5.2 for a graphical depiction of the variables and the clauses. Again, each solid triangle represents a 3-variable clause, and each dashed edge represents a 2-variable clause.

The first two clauses are $\text{NAE}(\mu_i^1, v_i, w_i) \wedge \text{NAE}(\hat{\mu}_i^2, v_i, w_i)$, each of weight 2^{13n}. Next we have clauses $(\hat{\mu}_i^2 \neq 1) \wedge (\mu_i^1 \neq 0)$ of weight 2^{12n}, followed by a clause $(\mu_i^2 \neq \hat{\mu}_i^2)$ of weight 2^{11n}. Then we have clauses $\text{NAE}(e_i, \mu_i^1, \mu_i^2)$ of weight 2^{10n}, and $e_i \neq 0$ of weight 2^{9n}. Finally, we have a clause $(\hat{e}_i \neq e_i)$, of weight 2^{8n}, that forces the variable $\hat{e}_i$ to be the negation of e_i.

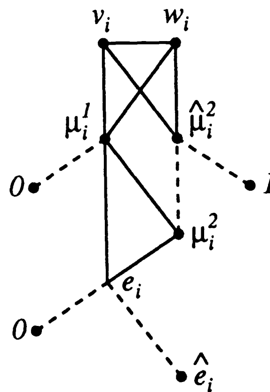


FIG. 5.2. *Gadget for type 7; triangles and dashed edges represent clauses.*

The next three types of clauses flip control variables y and z of the circuits C and T_i at the right moments, depending on the values of the variables d and e. Sometimes we want to flip the control variables to their unnatural setting (when we want to flip an input to a circuit), while at other times we want to flip them back to their natural settings (while correcting the gates of the circuit). These seemingly conflicting goals can be achieved by carefully synchronizing the changes to the variables d_i and e_i with the desired changes to the different control variables.

The type 8 clauses encourage setting the control variables of T_i to their unnatural value if $e_i = 1$: For each $i = 1, \cdots, p$ and for each $k = 1, \cdots, 2n$, we have clauses NAE $(e_i, y_{i,k}, 1)$ and NAE $(\hat{e}_i, z_{i,k}, 0)$ of weight 2^{7n} each.

The clauses of type 9 encourage setting the control variables of the circuit C and all test circuits T_r with $r \neq i$, to their natural value if $e_i = 1$: For each $i = 1, \cdots, p$, each $r = 1, \cdots, p$ with $r \neq i$ and each control variable of C and T_r, we have clauses

$$\text{NAE}\,(e_i, z_k, 1) \wedge \text{NAE}\,(e_i, z_{r,k}, 1) \wedge \text{NAE}\,(\hat{e}_i, y_k, 0) \wedge \text{NAE}\,(\hat{e}_i, y_{r,k}, 0),$$

each of weight 2^{6n}.

The clauses of type 10 encourage setting the control variables of C and of the circuits T_r with $r \neq i$ to their unnatural values, when $d_i = 1$. For each $i = 1, \cdots, p$, each $r = 1, \cdots, p$ with $r \neq i$, and each control variable of C and T_r, we have clauses

$$\text{NAE}\,(d_i, y_k, 1) \wedge \text{NAE}\,(d_i, y_{r,k}, 1) \wedge \text{NAE}\,(\hat{d}_i, z_i, 0) \wedge \text{NAE}\,(\hat{d}_i, z_{r,k}, 0)$$

of weight 2^{5n} each.

The final clauses of type 11 encourage the right default values for the variables, and give credit for having w_i be the negation of v_i. We have the following clauses, all of weight 2^{4n}: $(v_i \neq w_i)$, $(d_i \neq 1)$, $(z_k \neq 1)$ and $(z_{i,k} \neq 1)$, $(y_i \neq 0)$ and $(y_{i,k} \neq 0)$.

This concludes the definition of the POS NAE 3SAT instance I. In the following, we let A be a locally optimal assignment to the variables of I. We prove in a sequence of lemmas that A induces a locally optimal solution to the instance of FLIP; i.e., flipping any input variable v_i does not improve the output of the circuit C. Every lemma includes an implicit dependent clause: "If A is a local optimum of the instance I." The proofs of some lemmas are subdivided into simpler claims.

LEMMA 5.1. *If every d_i is* 0, *every $v_i = \neg w_i$, every gate is correct, and every $z_{i,k}$ is* 0, *then the assignment A induces a locally optimal solution to the* FLIP *instance.*

Proof. We claim that, if we could improve the output of the circuit C by flipping some v_i, then in A, we could get an improvement by flipping the corresponding variable d_i to 1. Let f be the weight in the FLIP instance of the assignment to the input variables of C induced by A, and suppose that flipping the variable v_i increases the weight to f'. Suppose that in A we flip the variable d_i to 1. First note that the clauses of types 1 and 2 are unaffected, because all other d variables are 0. Since all z variables for the T circuits are 0, no clauses of type 3 are lost. We claim that we gain in type 4 clauses. Since we have $v_i = \neg w_i$ in A, and all gates are correct, f is the sum of the numbers 2^j over all $j = 0, 1, \cdots, m$ for which $c_j = 1$, and f' is the sum of the numbers 2^j over all j for which $t_{i,j} = 1$, or equivalently, $\hat{t}_{i,j} = 0$. The type 4 clauses that contain d_i and are satisfied when $d_i = 0$ are: all clauses NAE $(d_i, \hat{t}_{i,j}, 1)$ and the clauses NAE $(d_i, c_j, 0)$ with $c_j = 1$, for a total weight of $2^{20n} \cdot (\sum_j 2^j + f)$. When $d_i = 1$, the type 4 clauses that contain d_i and are satisfied are: all clauses NAE $(d_i, c_j, 0)$ and those clauses NAE $(d_i, \hat{t}_{i,j}, 1)$ in which $\hat{t}_{i,j} = 0$; their total weight is $2^{20n} \cdot (\sum_j 2^j + f')$. □

The remaining task is to prove that in A, the long hypothesis of Lemma 5.1 holds.

LEMMA 5.2. *There is at most one i, $1 \leqq i \leqq p$ such that $d_i = 1$. For every i, the variable $\hat{d}_i$ is the negation of d_i.*

Proof. The proof relies on clauses of types 1 and 5. If $d_i = d_j = 1$ for some pair $i \neq j$, then we can gain a clause of type 1 by flipping d_i or d_j to 0. This establishes the first claim. Regarding the second claim, note that $\hat{d}_i$ occurs for the first time in the type 5 clauses. If $d_i = 1$, then we will set $\hat{d}_i$ to 0 to gain the clause NAE $(d_i, \hat{d}_i, 1)$. If $d_i = 0$, this clause is already satisfied, so we will set $\hat{d}_i$ to 1 to satisfy the next clause $(\hat{d}_i \neq 0)$. □

The next lemma characterizes the e variables. These are not mentioned in Lemma 5.1, but they play a central role in flipping the control variables of the circuits.

LEMMA 5.3. *For every $i = 1, \cdots, p$, the variable e_i is 1 if and only if $v_i = w_i$. The variable $\hat{e}_i$ is the negation of e_i.*

Proof. The proof relies on the clauses of type 7, the first clauses that contain e_i and $\hat{e}_i$. We need first a preliminary claim.

CLAIM 5.3A. *If $v_i = w_i$, then $\mu_i^1 \neq \mu_i^2$. If $v_i \neq w_i$, then $\mu_i^1 = \mu_i^2 = 1$.*

Proof. Suppose first that $v_i = w_i$. We will set the variables μ_i^1 and $\hat{\mu}_i^2$ to the negation of v_i and w_i to satisfy the heaviest clauses of type 7. Then, we will set μ_i^2 to the negation of $\hat{\mu}_i^2$, and thus of μ_i^1, to satisfy the clause $(\mu_i^2 \neq \hat{\mu}_i^2)$.

Suppose now that $v_i \neq w_i$. Then the heaviest type 7 clauses are already satisfied, so we will set $\mu_i^1 = 1$ and $\hat{\mu}_i^2 = 0$ to satisfy the next clauses. Then we will set μ_i^2 to the negation of $\hat{\mu}_i^2$, i.e., $\mu_i^2 = 1$. □

We can now prove the lemma. Suppose first that $v_i \neq w_i$. By the claim, both μ_i^1 and μ_i^2 are 1. We will set e_i to 0 to satisfy the clause NAE (μ_i^1, μ_i^2, e_i). If $v_i = w_i$, then $\mu_i^1 \neq \mu_i^2$ and this clause is already satisfied regardless of the value of e_i, so we will set $e_i = 1$ to satisfy the next clause of e_i. The variable $\hat{e}_i$ must be set to $\neg e_i$ to satisfy the clause $(\hat{e}_i \neq e_i)$. □

The next several lemmas summarize the properties of the clauses of types 2 and 3. They are similar to Lemmas 4.1, 4.2, and 4.3 of the previous section.

LEMMA 5.4. *If the gate g_i of the circuit C is incorrect, then $z_i = 1$. If $y_i = 0$, then $z_i = 1$. If $z_i = 1$, then for all $j > i$, $y_j = 0$, $z_j = 1$. Furthermore, if $y_{2n+1} = 0$, then some variable d_k is 1.*

Proof. Except for the last statement about a variable d_k, the lemma is identical to Lemma 4.1. The proof is also the same, and is omitted. We prove only the additional statement. The proof relies on the clauses of type 2C. If $z_{2n} = 1$, then we must have $y_{2n+1} = 0$ to satisfy the clause NAE $(z_{2n}, y_{2n+1}, 1)$. Therefore if all d variables are 0, then we will flip one of them to 1 to gain a clause NAE $(y_{2n+1}, d_k, 0)$. □

We have the analogous lemma for each test circuit T_r. In this case the lemma does not include a statement about d_r.

LEMMA 5.5. *The following hold for every circuit T_r. If gate $g_{r,i}$ is incorrect, then $z_{r,i} = 1$. If $y_{r,i} = 0$, then $z_{r,i} = 1$. If $z_{r,i} = 1$, then for all $j > i$, we have $y_{r,j} = 0$ and $z_{r,j} = 1$.* □

The following two lemmas are the analogues of Lemmas 4.2 and 4.3 for the circuits C and T_r. Their proofs are identical and are omitted.

LEMMA 5.6. (1) *If $y_{i-1} = 0$ and $z_{i-1} = 1$ for some even index i, then flipping either input of the gate i of the circuit C does not affect the type* 2A *clauses that correspond to gate i.*

(2) *Similarly, if for some circuit T_r and even index i we have $y_{r,i-1} = 0$ and $z_{r,i-1} = 1$, then flipping either input of the gate i of T_r does not affect the type* 3A *clauses that correspond to the gate i of T_r.* □

LEMMA 5.7. (1) *Suppose that $y_{i-1} = 1$ and $z_{i-1} = 0$ for some even index i. If the gate variable g_i is correct, then flipping z_i and y_i does not affect the type* 2A *clauses. If g_i is not correct, then flipping g_i would not affect the type* 2A *clauses corresponding to the gate i of circuit C and would gain a type* 2E *clause.*

(2) *Suppose that in some circuit T_r we have $y_{r,i-1}=1$ and $z_{r,i-1}=0$ for some even index i. If the gate variable $g_{r,i}$ is correct, then flipping $z_{r,i}$ and $y_{r,i}$ does not affect the type* 3A *clauses. If the gate variable $g_{r,i}$ is incorrect, then flipping $g_{r,i}$ would not affect the type* 3A *clauses corresponding to gate i of the circuit T_r and would gain a type* 3E *clause.* □

We can now show an analogue to Lemma 4.4. This lemma is not quite as immediate because its proof involves flipping gate variables and control variables that occur in other clauses besides types 2 and 3.

LEMMA 5.8. (1) *If all the d variables are* 0, *or if some e variable is* 1, *then all gate variables of C are correct and all control variables of C have their natural value* ($y_i=1$, $z_i=0$).

(2) *For every test circuit T_r, if $d_r=e_r=0$, then all gate variables of T_r are correct and all control variables have their natural value* ($y_{r,i}=1$, $z_{r,i}=0$).

Proof. We first prove statement (1). If all the d variables are 0, then (1) follows from Lemma 5.4. Thus we assume that some d variable is 1, which by Lemma 5.2 implies that exactly one d variable is 1. From the hypothesis, we may assume that some e variable is 1. The control variables of the circuit C occur in clauses of type 9, 10, and 11, besides the clauses of type 2. Since some e variable is 1, that variable pushes the control variables of C towards their natural values through type 9 clauses. These clauses play the role of the clauses of type C of the previous section.

As in the proof of Lemma 4.4, suppose that (1) does not hold, and let i be the smallest index such that z_i or y_i has an unnatural value, i.e., $z_i=1$ or $y_i=0$. We shall show first that if i is even, i.e., it corresponds to a gate g_i of C, then the gate variable g_i is correct. This variable occurs in clauses of type 2, and more specifically of types 2A and 2E. In addition, if g_i is an output gate, i.e., it is actually c_j for some j, then it occurs also in type 4 clauses NAE $(d_k, c_j, 0)$ of weight $2^{20n}\cdot 2^j$ for all $k=1,\cdots,p$, and in the clause $c_j\neq 1$ with weight $(p-1)\cdot 2^{20n}\cdot 2^j$. Since exactly one of the d variables is 1 and the other $p-1$ are 0, flipping c_j (i.e., the variable g_i) does not affect the total weight of satisfied type 4 clauses. By the previous two lemmas, if we flip g_i to its correct value, we do not lose any type 2A clauses, and we gain a clause of type 2E. Thus we conclude that if i is even, then g_i is correct.

If y_i is 0, then flipping it to 1 gains a type 9 clause. By Lemma 5.7 we do not lose any clauses of type 2A. Since $z_{i-1}=0$, we do not lose any type 2B clauses. If y_i belongs also to type 2C clauses, that is $i=2n+1$, then note that we gain $p-1$ clauses NAE $(y_{2n+1}, d_k, 0)$ of weight 2^{30n} each and lose the clause $(y_{2n+1}\neq 1)$ of weight $(p-1)\cdot 2^{30n}$; i.e., the net effect on the type 2C clauses is 0. Variable y_i belongs also to clauses of type 2D, but these have much smaller weight than type 9. Therefore, $y_i=1$. Similarly, if $z_i=1$, then flipping it to 0 gains a type 9 clause without losing any clauses of types 2A, 2B, or 2C.

The proof of statement (2) is very similar. Let T_r be a test circuit for which $d_r=e_r=0$. The control variables of T_r occur in clauses of type 8, 9, 10, and 11, besides the clauses of type 3. Since $d_r=e_r=0$, the type 8 and 10 clauses are already satisfied and have no effect on the control variables. The type 11 clauses push the control variables towards their natural values, while type 9 either are already satisfied (if all e variables are 0) or push them also in the same direction. These clauses play the role of the type C clauses of the previous section.

Suppose that (2) does not hold and let i be the smallest index for which $z_{r,i}=1$. Suppose first that i is even and the gate $g_{r,i}$ is incorrect. This variable occurs only in type 3 clauses, except if it is the negation gate $\hat{t}_{r,j}$ of an output, in which case it occurs also in a type 4 clause NAE $(d_r, \hat{t}_{r,j}, 1)$; in this case, the clause is satisfied since $d_r=0$ and thus has no effect. As above, we can argue then that the gate $g_{r,i}$ is correct. By a

similar argument as for part (1), we can show that if $y_{r,i}=0$, then we gain by flipping it to 1, and if $z_{r,i}=1$, then we gain by flipping $z_{r,i}$ to 0. In the case of $z_{r,2n}$ observe that we do not lose its type 3C clause NAE $(z_{r,2n}, d_r, 1)$ because $d_r=0$. □

LEMMA 5.9. *For every* $i=1,\cdots,p$, *if* $d_i=0$, *then* $e_i=0$ *and* $v_i=\neg w_i$.

Proof. From Lemma 5.3, $e_i=1$ if and only if $v_i=w_i$. We assume that $e_i=1$ and $d_i=0$. Suppose that we flip w_i (to $\neg v_i$) thereby gaining a clause of type 11. We prove that no heavier clauses are lost. Variable w_i belongs to clauses of type 3 as an input of the circuit T_i, and to clauses of types 6 and 7.

CLAIM 5.9A. *If* $e_i=1$ *and* $d_i=0$, *then all the control variables of the circuit* T_i *have their unnatural values*; *i.e.*, $y_{i,j}=0$ *and* $z_{i,j}=1$.

Proof. Since $e_i=1$, we can gain clauses of type 8 by flipping the control variables of T_i to their unnatural value. The only heavier clauses in which the control variables participate are the clauses of type 3. We flip their value in order of decreasing index. That is, first we flip $z_{i,2n}$ to 1. We do not lose any clause of type 3A because all clauses of type A that contain a z variable also contain a 0. Also, we do not lose the only clause NAE $(y_{i,2n}, z_{i,2n}, 0)$ of type 3B that contains $z_{i,2n}$, and we do not lose the clause NAE $(z_{i,2n}, d_i, 1)$ of type 3C because $d_i=0$. Thus we conclude that if $z_{i,2n}=0$, then we can improve the assignment by flipping this variable to 1. Arguing in the same manner for the rest of the control variables in the order $y_{i,2n}, z_{i,2n-1}, y_{i,2n-1}, \cdots, z_{i,1}, y_{i,1}$, we can conclude that they all have their unnatural values. □

The next claim concerns the type 6 clauses.

CLAIM 5.9B. *If* $d_i=0$, *then flipping* w_i *does not affect the type* 6 *clauses.*

Proof. Suppose that $\eta_i=w_i$. Then flipping w_i to be the opposite of η_i cannot cause the loss of the type 6 clauses that contain w_i. Thus assume that $\eta_i\neq w_i$. Since $d_i=0$, and thus $\hat{d}_i=1$ by Lemma 5.2, the first two clauses of type 6 are satisfied regardless of the value of θ_i^1 and θ_i^2. The following two clauses are also satisfied because $\eta_i\neq w_i$. Thus we may assume that $\theta_i^1=\theta_i^2=\neg\eta_i$, so that the clauses $(\theta_i^1\neq\eta_i)$ and $(\theta_i^2\neq\eta_i)$ are satisfied. Therefore the type 6 clauses containing w_i are satisfied regardless of its value. □

The variable w_i occurs in clauses of type 3A as the input of some gates $g_{i,j}$ of the circuit T_i. Since we gave only even indices to the gates, $j\geqq 2$, and by Claim 5.9A, $z_{i,j-1}=1$ and $y_{i,j-1}=0$. It follows then from Lemma 5.6 that flipping w_i does not affect these clauses. By Claim 5.9B, it does not affect the clauses of type 6 either. The clauses of type 7 that contain w_i also contain v_i; if $v_i=w_i$, then we cannot lose these clauses by flipping w_i. It follows from Lemma 5.3 that $e_i=0$. □

The final lemma shows that all the d variables are 0.

LEMMA 5.10. *Every* $d_i=0$.

Proof. We prove a sequence of claims that restrict A under the assumption $d_i=1$, eventually leading to a contradiction. The sequence of claims follows the sequence of flips that take place after we flip d_i to 1 (recall the description that we gave before the definition of the instance I).

CLAIM 5.10A. *If* $d_i=1$ *and* $v_i=\neg w_i$, *then all the control variables of the circuit* C *and the circuits* T_r *with* $r\neq i$ *have their unnatural value*: $z_k=z_{r,k}=1$ *and* $y_k=y_{r,k}=0$ *for all* k.

Proof. Since $d_i=1$ we can gain clauses of type 10 by flipping to their unnatural values all the control variables except for the ones of the circuit T_i. As in the preceding lemma, these variables can be flipped in decreasing order of the subscript k, so as not to lose clauses of types 2B or 3B. The clauses of types 2A and 3A are never lost by flipping a z to 1 or a y to 0.

When we flip the variable y_{2n+1} to 0 we gain one clause of type 2C and lose $p-1$ clauses of type 2C, but their weights exactly cancel. Because $v_i=\neg w_i$, we know that

$e_i = 0$ by Lemma 5.3. Since $d_i = 1$, we have that all the other d variables are 0 by Lemma 5.2. It follows from Lemma 5.9 that all the variables e_r, for $r \neq i$ are also 0. Thus flipping the control variables does not lose any clauses of types 8 or 9. Types 1, 4, 5, 6, 7 do not involve the variables that are flipped. The only clauses of type 3 that we could lose are those involving a d variable that is 1. By Lemma 5.2, only d_i is 1, and the type 3 clause involving d_i is not affected since the claim explicitly excludes the variables of the circuit T_i. □

We shall show that if $v_i \neq w_i$, then we can gain a type 6 clause by flipping v_i. First, we need the following claim concerning the local variables of type 6.

CLAIM 5.10B. *If* $d_i = 1$, *then* $\eta_i \neq w_i$.

Proof. Since $d_i = 1$, and thus $\hat{d}_i = 0$, we can gain the heaviest clauses of type 6 by flipping θ_i^1 to 0 and θ_i^2 to 1. If $\eta_i = w_i$, then one of the clauses NAE $(\theta_i^1, \eta_i, w_i)$, NAE $(\theta_i^2, \eta_i, w_i)$ is not satisfied, but we can satisfy both of them by flipping η_i to be the negation of w_i. □

CLAIM 5.10C. *If* $d_i = 1$, *then* $v_i = w_i$.

Proof. Suppose $v_i = \neg w_i$. If we flip v_i to agree with w_i, then by Claim 5.10B we gain the type 6 clause $(\eta_i \neq v_i)$. The variable v_i also occurs in type 2A clauses as an input to gates of the circuit C and in type 3A clauses as an input to gates of the circuits T_r with $r \neq i$. Recall that in T_i we use the variable w_i as an input in place of v_i. If v_i is an input to a gate g_j or $g_{r,j}$, then $j \geqq 2$ and by Claim 5.10A the corresponding control variables with index $j-1$ exist and have their unnatural value. By Lemma 5.6, flipping v_i does not affect the type 2 and type 3 clauses in which it occurs. Since v_i does not occur in any other clauses higher than type 6, the claim follows. □

CLAIM 5.10D. *If* $d_i = 1$, *then all the gates of the circuit* C *are correct, and all the control variables of* C *have their natural values.*

Proof. By Claim 5.10C, $v_i = w_i$. By Lemma 5.3, $e_i = 1$. The claim now follows from Lemma 5.8. □

CLAIM 5.10E. *If* $d_i = 1$, *then all gates of the circuit* T_i *are correct.*

Proof. Suppose that gate k is incorrect in the circuit T_i. By Lemma 5.5, $z_{i,q} = 1$ for every $q > k$. Thus we can gain the clause NAE $(z_{i,2n}, d_i, 1)$ of type 3C by flipping d_i to 0. We never lose any type 1 clauses by flipping d_i to 0 because they all have a 1 in them. Also, since $y_{2n+1} = 1$ by Claim 5.10D, we do not lose a type 2 clause. □

Now we can complete the proof of the lemma. We can gain a clause of type 11 by flipping d_i to 0. As we mentioned above, we do not lose any clauses of type 1 or 2. Except for the type 4 clauses, all other NAE clauses that contain d_i (types 3, 5, 6, 10) also contain the constant 1; thus they cannot be lost by setting $d_i = 0$. Regarding the type 4 clauses, we know from Claims 5.10D and 5.10E that all the gates of the circuits C and T_i are correct. The fact that $v_i = w_i$ implies that $c_j \neq \hat{t}_{i,j}$, for all j. Thus the weight of the clauses of type 4 that we lose by flipping d_i to 0 is exactly equal to the weight of the clauses that we gain. □

We can now prove that a locally optimal assignment A to our POS NAE 3SAT instance I induces a locally optimal solution to the FLIP instance C. By Lemma 5.10, all d variables are 0. Lemma 5.9 implies that all e variables are 0, which by Lemma 5.3 implies that $v_i = \neg w_i$ for all i. Furthermore, by Lemma 5.8, all the gates in all the circuits C and T_i are correct, and all the control variables have their natural value. The result follows from Lemma 5.1. Thus we have shown Theorem 5.11.

THEOREM 5.11. POS NAE 3SAT *is* PLS-*complete.* □

Combining this with the reductions of § 3 we get Corollary 5.12.

COROLLARY 5.12. MAX CUT 2SAT, SWAP, FM-SWAP, FM-GRAPH PARTITIONING, *and* STABLE CONFIGURATION *are* PLS-*complete.* □

The result for SWAP solves one of the main open problems in [9]. As we noted

above, any locally optimal solution for KL-GRAPH PARTITIONING is locally optimal for the same instance of SWAP. Thus we have given an alternate proof of the result of [9] that KL-GRAPH PARTITIONING is PLS-complete. The result for STABLE CONFIGURATION addresses a problem raised by Haken, Luby, Godbeer, Lipscomb, and Parberry [5], [4], [17], [19].

Next we prove that the standard local search algorithm has exponential running time in the worst case, by proving that our reduction to POS NAE 3SAT is actually a tight reduction. We assume without loss of generality that the circuit C of the FLIP instance contains gates that compute the negations of the input variables (i.e., NOR (v_i, v_i) for all i). Thus if the gates of C have their correct values and we flip any input variable v_i, then some gate will become incorrect. Of course, if C does not have such gates, we can add them before applying the reduction.

We define the set $\mathcal{R}$ of *reasonable* assignments for our instance I to be the set of assignments that (1) satisfy a maximum-weight set of clauses from types 1, 2A, 2B, 2C, 3A, 3B, 3C (we refer to these as the *heavy* clauses), and (2) have $d_i = 0$ for all i. Note that we cannot satisfy simultaneously all the heavy clauses; in particular, if we satisfy the type 1 clauses (at most one d_i variable is 1), then we cannot satisfy all the type 2C clauses: NAE $(y_{2n+1}, d_i, 0)$ of weight 2^{30n} for all $i = 1, \cdots, p$ and $(y_{2n+1} \neq 1)$ of weight $(p-1) \cdot 2^{30n}$. However, we can satisfy all the other clauses. That is, condition (1) above for an assignment to be reasonable can be stated equivalently as: the assignment satisfies all the heavy clauses except for a weight of $(p-1) \cdot 2^{30n}$. An important observation is that if an assignment satisfies a maximum-weight set of heavy clauses, then it must have the properties stated in Lemmas 5.4 and 5.5, because otherwise we can gain a heavy clause by some variable flip.

We argue that our choice of $\mathcal{R}$ satisfies the conditions for the reduction to be tight (see Definition 3.2). First, as we proved, $\mathcal{R}$ contains all locally optimal assignments of I. Second, for any solution of the FLIP instance, i.e., assignment V to the input variables $v_1, \cdots, v_p$, we can construct a corresponding reasonable assignment A to the variables of I that agrees with V as follows. We let all $d_i = 0$, $v_i = \neg w_i$, give the correct value to all the gate variables of the circuit C and the circuits T_i, and set all the control variables to their natural values. Since the gate variables have the correct values, we can satisfy all the clauses of types 2A and 3A by giving appropriate values to the local variables.

Consider now the transition graph of I, TG(I), and suppose that there is a (weight-improving) path from one reasonable assignment A to another reasonable assignment A' so that all the intermediate vertices of the path (if there are any) are outside $\mathcal{R}$. Let V and V' be the restrictions of the assignments to the input variables v_i. We have to show that either $V = V'$ or V' is obtained from V by flipping a single input variable, and that in the latter case V' is a better solution to the FLIP instance than V.

LEMMA 5.13. *Suppose that $V' \neq V$. Then V' is obtained from V by flipping a single input variable v_i. Furthermore, the path in* TG(I) *from A to A' is nontrivial* (*has more than one arc*) *and starts by flipping the corresponding variable d_i.*

Proof. Assume first that the path consists of a single arc, i.e., A' is obtained from A by flipping a single variable. If that variable is not one of the v_i's, then $V = V'$. If we flip a v_i, then some gate of C becomes violated; since all the d variables are 0, this implies (by Lemma 5.4) that we lose a heavy clause. That is, A' is not reasonable and in fact is worse than A, a contradiction.

We conclude that the path from A to A' is nontrivial. The clause weights are scaled so that if an assignment satisfies a maximum-weight set of heavy clauses, then so does any better assignment. Thus all vertices of the path satisfy condition (1) above,

and therefore, do not satisfy condition (2). Since the type 1 clauses ensure that at most one d variable is 1 (Lemma 5.2), it follows that we move from A to the next vertex of the path by flipping some variable d_i to 1, which stays at this value until we reach A', at which point we flip it back to 0. Since $d_i = 1$ in all the vertices of the path, the control variable $z_{i,2n}$ corresponding to the last gate of T_i must be 0, because otherwise we would lose the type 3C clause NAE $(z_{i,2n}, d_i, 1)$. By Lemma 5.5 this implies that all gate variables of T_i are correct during the path. If we flip some variable v_r with $r \neq i$ somewhere along the path, we would make some gate of T_i incorrect. Therefore either $V = V'$ or V' is obtained from V by flipping v_i. □

LEMMA 5.14. *If $V' \neq V$, then V' is a better solution to the* FLIP *instance than V.*

Proof. Consider the flip of d_i to 1 in going from A to the next vertex along the path. Since A is reasonable, flipping d_i cannot gain a heavy clause. Variable d_i occurs also in clauses of types 4, 5, 6, 10, 11. Observe that, except for the type 4 clauses, all other NAE clauses containing d_i contain also a 1; thus flipping d_i to 1 cannot gain any clauses except possibly for type 4. Since the assignment A satisfies condition (1) and has all d variables equal to 0, all gate variables of C must be correct in A; thus the variables c_j reflect correctly the weight of V in the FLIP instance. Since the assignment obtained from A by flipping d_i to 1 satisfies also condition (1), all gate variables of T_i must be correct. If $w_i = v_i$, then T_i has the same inputs as C, and thus $\hat{t}_{i,j} = \neg c_j$ for all j, which implies that flipping d_i neither gains nor loses weight among the type 4 clauses. Therefore, we must have $w_i = \neg v_i$, and thus the negations of the variables $\hat{t}_{i,j}$ reflect correctly the weight of V' in the FLIP instance. Since flipping d_i improves the assignment A, we conclude that V' is better than V in the FLIP instance C. □

Thus our reduction to POS NAE 3SAT is tight. Consequently, Lemma 3.3 and the reductions of § 3 imply Theorem 5.15.

THEOREM 5.15. *The following hold for the problems* POS NAE 3SAT, MAX CUT, 2SAT, SWAP, FM-SWAP, *and* STABLE CONFIGURATION.

(1) *The standard algorithm takes exponential time in the worst case.*

(2) *The standard algorithm problem is* NP-*hard.* □

Haken and Luby [5] proved an exponential bound on the running time for the STABLE CONFIGURATION problem in the case that the standard algorithm uses the steepest descent rule to choose a better neighbor. Our theorem holds for *any* rule, even a nondeterministic rule. The instances that are derived from the reductions have the property that starting from some initial configurations, all weight-improving paths to a stable configuration have exponential length. Haken has independently constructed instances with the same property [5a].

6. Conclusions. We have analyzed the complexity of several natural, simple local search problems: MAX CUT, 2SAT, GRAPH PARTITIONING under the single SWAP and FM-SWAP neighborhoods, and finding a STABLE CONFIGURATION in connectionist networks. We showed that these problems are PLS-complete in the general weighted case, and are P-complete in the unweighted case. Since these problems are quite restricted and the neighborhoods are so simple, it appears likely that we could find reductions to many other problems to show similar results, and thus, that the class of PLS-complete problems is actually quite extensive.

Consider, for example, the WEIGHTED INDEPENDENT SET problem: Given a graph with weights on the vertices, find an independent set of maximum weight. It was noted in [9] that we can find, by a simple greedy (polynomial-time) algorithm, a local optimum under the following "single swap neighborhood": the neighbors of an independent set I are the sets obtained by adding to I a vertex v and removing from

I the vertices that are adjacent to I. Note that in the unweighted case the locally optimal independent sets under this neighborhood are exactly the maximal independent sets (and thus, we can obtain one in NC). Consider now the following 2-step neighborhood: if after adding a vertex v to the independent set I and removing its adjacent vertices the new set I' is not maximal, then augment it (arbitrarily) to a maximal independent set. We can show by applying the standard NP-completeness reduction from 3SAT (or even SAT) that finding a locally optimal independent set under this 2-step neighborhood is PLS-complete in the general case, and P-complete in the unweighted case, and this holds regardless of how we choose to augment in the second step if there is more than one choice.

In most cases, the known NP-completeness reductions do not work and we must use much more delicate and complicated constructions to achieve a PLS-reduction, as undoubtedly the reader has noticed from the proofs of the last two sections. In the case of the TRAVELING SALESMAN Problem, Papadimitriou has recently found a reduction from 2SAT to the TSP under the Lin–Kernighan neighborhood, thereby showing the latter problem PLS-complete. Krentel recently has shown PLS-completeness for the k-OPT neighborhood (by a reduction from FLIP), if k is a sufficiently large constant [14]. An interesting open problem is to determine whether the same result holds for the simple 2-OPT or 3-OPT neighborhoods.

Acknowledgments. The first author thanks Mark Krentel for suggesting this problem area and for providing lots of encouragement as these results were slowly discovered. The second author thanks Vijay Vazirani for helpful discussions.

Note added in proof. The standard algorithm problem for all the local search problems considered in this paper is PSPACE-complete. A proof of this for the FLIP problem is given in [18a, Lemma 4]; completeness for the other problems follows then from the tightness of the reductions.

REFERENCES

[1] J. Bruck and J. W. Goodman, *A generalized convergence theorem for neural networks*, IEEE Trans. Inform. Theory, 34 (1988), pp. 1089–1092.

[2] A. E. Dunlop and B. W. Kernighan, *A procedure for placement of standard-cell* VLSI *circuits*, IEEE Trans. Computer-Aided Design, 4 (1985), pp. 92–98.

[3] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proc. 19th Annual Design Automation Conference, 1982, pp. 175–181.

[4] G. Godbeer, *On the computational complexity of the stable configuration problem for connectionist models*, Master's thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, September, 1987.

[5] A. Haken and M. Luby, *Steepest descent can take exponential time for symmetric connection networks*, Complex Systems, 2 (1988), pp. 191–196.

[5a] A. Haken, *Connectionist networks that need exponential time to stablize*, manuscript.

[6] J. J. Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proc. Nat. Acad. Sci. U.S.A., 79 (1982), pp. 2554–2558.

[7] J. J. Hopfield and D. W. Tank, *Neural computation of decisions in optimization problems*, Biol. Cybernet., 52 (1985), pp. 141–152.

[8] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, *Optimization by simulated annealing: an experimental evaluation, Part* I (*graph partitioning*), Oper. Res., to appear.

[9] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, *How Easy Is Local Search*? (extended abstract), in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 39–42; J. Comput. System Sci., 37 (1988), pp. 79–100.

[10] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a perfect matching is in random* NC, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1985, pp. 22-32; Combinatorica, 6 (1986), pp. 35-48.

[11] R. M. KARP AND A. WIGDERSON, *A fast parallel algorithm for the maximal independent set problem*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 266-272; J. Assoc. Comput. Mach., 32 (1985), pp. 762-773.

[12] B. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Tech. J., 49 (1970), pp. 291-307.

[13] M. W. KRENTEL, *On finding locally optimal solutions*, in Proc. 4th Annual IEEE Conference on Structure in Complexity, Eugene, OR, 1989, pp. 132-137; SIAM J. Comput., 19 (1990), pp. 742-749.

[14] ———, *Structure in locally optimal solutions*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989.

[15] R. LADNER, *The circuit value problem is log space complete for* P, SIGACT News, 7 : 1 (1975), pp. 18-20.

[16] S. LIN AND B. KERNIGHAN, *An effective heuristic for the traveling salesman problem*, Oper. Res., 21 (1973), pp. 498-516.

[17] J. LIPSCOMB, *On the computational complexity of finding a connectionist model's stable state of vectors*, Master's thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, October, 1987.

[18] M. LUBY, *A simple parallel algorithm for the maximal independent set problem* (extended abstract), in Proc. 17th Annual ACM Symposium on Theory of Computing, IEEE Computer Society, Washington, DC, 1985, pp. 1-10; SIAM J. Comput., 15 (1986), pp. 1036-1053.

[18a] C. H. PAPADIMITRIOU, A. A. SCHÄFFER, AND M. YANNAKAKIS, *On the complexity of local search*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 438-445.

[19] I. PARBERRY, *A primer on the complexity theory of neural networks*, in A Sourcebook on Formal Techniques in Artificial Intelligence, R. B. Banerji, ed., Elsevier, Amsterdam, the Netherlands, 1989.

[20] A. SCHRIJVER, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 88-99, February 1991

© 1991 Society for Industrial and Applied Mathematics
005

IMPROVED UPPER AND LOWER TIME BOUNDS FOR PARALLEL RANDOM ACCESS MACHINES WITHOUT SIMULTANEOUS WRITES*

IAN PARBERRY† AND PEI YUAN YAN‡

Abstract. The time required by a variant of the PRAM (a parallel machine model which consists of sequential processors which communicate by reading and writing into a common shared memory) to compute a certain class of functions called *critical* functions (which include the Boolean OR of n bits) is studied. Simultaneous reads from individual cells of the shared memory are permitted, but simultaneous writes are not. It is shown that any PRAM which computes a critical function must take at least $0.5 \log n - O(1)$ steps, and that there exists a critical function which can be computed in $0.57 \log n + O(1)$ steps. These bounds represent an improvement in the constant factor over those previously known.

Key words. CREW PRAM, critical function, time, lower bound, upper bound

AMS(MOS) subject classifications. 68Q05, 68Q10, 68Q20, 68Q25

1. Introduction. A PRAM consists of an infinite collection of sequential processors connected to a common shared memory. In each time interval, or step, each processor reads a value from the shared memory, moves into a new state, and writes this new state back into the shared memory. Simultaneous reads of a single cell in the shared memory by many processors are permitted, but simultaneous writes are not. Computation of a function on n inputs proceeds by placing the n input values into the first n cells of the shared memory, starting the PRAM with all processors in a predefined initial state, and waiting for all processors to halt. When the computation is over, the output can be found in the first shared-memory cell. The running time of the PRAM is defined to be the number of steps taken, expressed as a function of n.

The lower bound techniques discussed in this paper are based on a communication argument, and thus hold even if the processors have infinite computational power. Many of the "standard" models which have become popular in the recent literature (for example, in Fortune and Wyllie [3], Goldschlager [4], [5], and Shiloach and Vishkin [11]) limit the local computational power of the individual processors. More details on the effect that this can have on the computational ability of PRAMs can be found in Parberry [6]-[8]. Our upper bound is intended to illustrate the limits of the lower bound technique by taking advantage of the unlimited computing power of the processors, and is not intended as a practical algorithm. We will, however, use a number of processors which is only a polynomial in the size of the input.

A Boolean function is said to be *critical* if there exists an input I with the property that changing any single bit of I changes its output. One example of a critical function is the Boolean OR of n bits (consider the all-zero input). The obvious parallel algorithm for computing the Boolean OR of n bits uses "successive doubling" and takes time $\lceil \log_2 n \rceil$. (From this point on, all logarithms will be to base two unless otherwise

* Received by the editors November 3, 1987; accepted for publication (in revised form) February 15, 1990. A preliminary version of the results in the paper appeared as part of the second author's Ph.D. dissertation, Department of Mathematics, Pennsylvania State University, University Park, Pennsylvania, 1989, and in the Proceedings of the 1989 International Conference on Parallel Processing, St. Charles, Illinois, August 1989.

† Department of Computer Science, 333 Whitmore Laboratory, Pennsylvania State University, University Park, Pennsylvania 16802. Present address, Department of Computer Science, University of North Texas, Denton, Texas 76203.

‡ Department of Mathematics, 407 McAllister Building, Pennsylvania State University, University Park, Pennsylvania 16802.

indicated.) This intuitively seems to be a lower bound, based on the number of bits that a processor can "know about" at each point in time. Independently, Cook and Dwork [2] and Reischuk [10] noticed that this "obvious" lower bound is incorrect. These authors combined their results in [1]. From this reference we learn that any PRAM which computes a critical function on n inputs must take time at least $\log_b n$ where $b = 0.5(5+\sqrt{21})$ is slightly greater than 4.79. We will show that it must take time at least $\log_4 n - O(1)$. From it we also learn that there is a critical function that can be computed by a PRAM in time $\log_3 n + 1$. We will show that there is a critical function which can be computed by a PRAM in time $\log_c n + O(1)$ where $c = 2+\sqrt{2}$ is slightly greater than 3.41. The relationship between these results is summarized in Tables 1.1 and 1.2. Other lower bounds for PRAMs without simultaneous writes can be found in Simon [12] and Snir [13].

The remainder of the paper is divided into four sections. The first describes the PRAM model in more detail, while the second contains some preliminary results concerning exclusive-write PRAMs. The lower and upper bounds are contained in the third and fourth sections, respectively.

2. The PRAM model. A PRAM consists of an infinite number of processors and an infinite shared memory. The shared memory is divided into *cells*, each of which is capable of holding a natural number (for the purposes of this paper, we count zero as being a natural number). Each processor has a *state*, which is also a natural number. The cells and the processors are numbered consecutively from zero. More formally, a PRAM M consists of a triple (r, s, w), where r, $w : \mathbf{N}^3 \to \mathbf{N}$, and $s : \mathbf{N}^4 \to \mathbf{N}$. An input to M consists of a finite sequence of natural numbers $I = (x_0, x_1, \cdots, x_{n-1})$. Each x_i is called an *input symbol*. At the start of the computation, x_i is placed into cell i of the shared memory, $0 \leqq i < n$, while cell i for $i \geqq n$ is set to zero. Each processor is placed in state zero. The computation on input I proceeds in a sequence of discrete *steps*. During the tth step, $t \geqq 1$, each processor p, $p \geqq 0$ does the following simultaneously. Suppose that p was in state $q \in \mathbf{N}$ at the end of the $(t-1)$th step (where the zeroth step refers to a point in time immediately before the computation began).

TABLE 1.1

Previous best-known upper and lower bounds on the time required for a PRAM *to compute critical functions. Each entry in the table shows the constant c where the dominant term in the bounds is of the form c* log *n.*

	Lower bound	Upper bound
Previous	0.44	0.64
Current	0.5	0.57

TABLE 1.2

Previous best-known upper and lower bounds on the number of input symbols of a critical function that a PRAM *can compute in t steps. Each entry in the table shows the constant c where the dominant term in the bounds is a constant times c^t.*

	Lower bound	Upper bound
Previous	3	4.79
Current	3.41	4

(1) Read a value v from shared memory cell $r(p, t, q)$.
(2) Compute $v' = s(p, t, q, v)$. The state of p at the end of the tth step is v'.
(3) Write the value v' into cell $w(p, t, v')$.

Simultaneous reads of a single shared-memory cell by many processors are permitted, but simultaneous writes into a single cell are not. The output of M will be found in shared memory cell zero when the processors have terminated. The *time* required by M is the maximum over all inputs of size n of the number of steps needed to process that input, expressed as a function of n.

For the purposes of our lower bounds, we will be more lenient than is customary in our definition of what it means from a PRAM to compute a function, in that we will allow some preprocessing of the inputs before the computation begins, and postprocessing of the output after the computation ends. We say that a PRAM *computes* $f: \mathbf{N}^* \to \mathbf{N}^*$ if there exist functions $\alpha, \beta: \mathbf{N} \to \mathbf{N}$ such that the PRAM on input $\alpha(x_0)$, $\alpha(x_1), \cdots, \alpha(x_{n-1})$ produces output $y \in \mathbf{N}$ where $\beta(y) = f(x_0, \cdots, x_n)$. For our upper bounds we will insist that no extra pre- or postprocessing can be done, that is, α and β are the identity functions.

We will say that two PRAMs are *functionally equivalent* if they compute the same function, and that they are *equivalent* if they are functionally equivalent and have the same running time.

3. Persistence and predictability in PRAMs.

DEFINITION 3.1. Let $I = (x_0, \cdots, x_{n-1}) \in \{0, 1\}^n$ be an input string. Define $I(u) = (x_0, \cdots, x_{u-1}, \bar{x}_u, x_{u+1}, \cdots, x_{n-1})$, where $\bar{0} = 1$ and $\bar{1} = 0$, to be the input which is identical to I except in the uth bit.

DEFINITION 3.2 (Cook, Dwork, and Reischuk [1]). If $0 \leqq u < n$, index u *affects* shared-memory cell c at time t on input I if the contents of c at time t is different on inputs I and $I(u)$.

DEFINITION 3.3 (Cook, Dwork, and Reischuk [1]). If $0 \leqq u < n$, index u *affects* processor p at time t on input I if the state of p at time t is different on inputs I and $I(u)$.

DEFINITION 3.4. A PRAM is called *persistent* if the following two properties hold:

(1) If index u affects processor p at time $t-1$ on input I, then index u affects processor p at time t on input I.

(2) If index u affects cell c at time $t-1$ on input I and processor p reads cell c at time t on input $I(u)$, then index u affects processor p at time t on input I.

The following two definitions are mutually recursive.

DEFINITION 3.5. If $0 \leqq u < n$, index u *counsels* shared-memory cell c at time t on input I if either $t = 0$ and c is the ith shared-memory cell, or $t > 0$ and either:

(1) No processor writes into c on input I at time t and either:
  (i) u counsels c at time $t-1$ on input I, or
  (ii) There exists a processor p which writes into c at time t on input $I(u)$, or
(2) Some processor p writes into cell c at time t on input I, and either:
  (i) No processor writes into cell c at time t on input $I(u)$, or
  (ii) Some processor p' writes into cell c at time t on input $I(u)$ and either:
    (a) $p' \neq p$, or
    (b) $p' = p$ and u counsels processor p at time t on input I.

DEFINITION 3.6. For $0 \leqq u < n$, index u *counsels* processor p at time t on input I if either:

(1) $t > 1$ and u counsels p at time $t-1$ on input I, or

(2) $t \geqq 1$, u counsels some cell c at time $t-1$ on input I and p reads c at time t on input I.

The definition of *counseling* is weaker than that of *affecting* in the sense that the only way that an index u can affect a cell or a processor at time t on input I is to counsel that cell or processor at time t on input I. The converse is not necessarily the case (that is, not all counseling leads to an affectation) since, for example, in Definition 3.5(1)(ii), processor p may at time t (fortuitously) write into cell c the same value that it contained at time $t-1$.

DEFINITION 3.7. A PRAM is called *predictable* if

(i) It is persistent, and

(ii) Every index u which counsels a cell or processor at time t on input I also affects that cell or processor at time t on input I, for all inputs I and $t \geqq 1$, and

(iii) If the state of processor p_1 at time t_1 on input I_1 is the same as the state of processor p_2 at time t_2 on input I_2, then $t_1 = t_2$ and $p_1 = p_2$.

We can conclude from the observation preceding Definition 3.7 that not every PRAM is predictable. However, the following result will enable us to take advantage of the extra structure which is present in a predictable PRAM.

LEMMA 3.8. *For every* PRAM *there exists an equivalent predictable* PRAM.

Proof. Let M be a $T(n)$ time-bounded PRAM. Define a new PRAM M' as follows. M' simulates M, with processor p of M' at time t simulating processor p of M at time t, subject to the following modifications. Let $p_0 = 2$ and for $i \geqq 1$, p_i be the smallest prime number exceeding p_{i-1}.

(1) Suppose processor p of M' was in state q at time $t-1$. If $q \neq 0$ then it factors q into a product of prime powers:

$$Q = p_0^{t-1} p_1^{p} \prod_{i=1}^{t-1} p_{i+1}^{q_i},$$

where $q_i \in \mathbf{N}$ for $1 \leqq i \leqq t-1$. The state of processor p of M at time $t-1$ can be computed from the sequence of values that it read during the first $t-1$ steps of the computation. The sequence $q_1, \cdots, q_{t-1}$ is used for that purpose. If $q = 0$, then the state of processor p of M is taken to be zero.

(2) Using the state of processor p of M at time $t-1$ determined in step (1), it ascertains which cell in the shared memory that processor p of M will read from at time t, and reads a value v from that cell. It factors v into a product of prime powers:

$$v = p_0^{t'} p_1^{p'} \prod_{i=1}^{t'} p_{i+1}^{v'_i},$$

where $v'_i \in \mathbf{N}$ for $1 \leqq i \leqq t-1$. The value v was written there by processor p' at time t'. The value that processor p' of M would have written into that cell can be obtained by the sequence of values read by processor p' of M in the first t' steps of the computation. The sequence of values $v'_1, \cdots, v'_{t'}$ is used for that purpose.

(3) Using the state of processor p of M at time $t-1$ determined in (1), and the value which processor p of M read from the shared memory at time t determined in (2), it computes the place in the shared memory into which the new state w of processor p of M at time t should be written.

(4) Finally, it computes its own new state $2qp_{t+1}^{v}$ and writes this into the shared memory in place of w.

The correctness of the simulation can be verified by induction on t. Some extra preprocessing of the inputs to M is necessary before they can be presented to M'. Each input symbol x should be replaced by 5^x. Some postprocessing of the output of M' is also necessary to obtain the output of M. The output of M' is to be decomposed

into a product of prime powers:

$$p_0^{T(n)} p_1^p \prod_{i=1}^{T(n)} p_{i+1}^{q_i},$$

where $q_i \in \mathbf{N}$ for $1 \leqq i \leqq t-1$. The output of M is then $q_{T(n)}$.

We claim that M' is predictable. Condition (1) of Definition 3.7 requires that M' be persistent. Suppose index u affects processor p of M' at time $t-1$ on input I. Then state q is different on inputs I and $I(u)$ in (1) above, which implies that the new state $2qp_{t+1}^v$ of processor p of M' computed in (4) above is also different on inputs I and $I(u)$, that is, u affects processor p of M' at time t on input I. Now suppose that index u affects cell c of M' at time $t-1$ on input I and that processor p of M' reads cell c at time t on input $I(u)$. There are two cases to consider.

Case 1. Processor p does not read cell c at time t on input I. Since it *does* read from there on input $I(u)$, it follows that u must affect processor p at time t on input I.

Case 2. Processor p reads cell c at time t on input I. The value v read in (2) above must be different on I and $I(u)$. Thus the new state $2qp_{t+1}^v$ computed in (4) must differ on I and $I(u)$, that is, index u affects processor p at time t on input I. Thus we conclude that M' is persistent.

Next we must demonstrate that property (ii) of Definition 3.7 holds, that is, for all t and I, if index u counsels shared-memory cell c or processor p at time t on input I, then u affects shared-memory cell c or processor p, respectively, at time t on input I. The proof is by induction on t. The hypothesis is certainly true for $t=0$. Now suppose that $t>0$ and that the hypothesis is true at time $t-1$. Let I be an arbitrary input, and u an arbitrary index.

Case 1. Let p be a processor, and suppose that index u counsels p at time t on input I. Then by Definition 3.6, either of the following two cases applies.

Case 1.1. u counsels p at time $t-1$ on input I, which implies by the induction hypothesis that u affects p at time $t-1$ on input I. Since M' is persistent, it follows that u affects p at time t on input I.

Case 1.2. u counsels some cell c at time $t-1$ on input I and p reads cell c at time t on input I. By the induction hypothesis, u affects cell c at time $t-1$ on input I. Since M' is persistent, this implies that u affects p at time t on input I.

Case 2. Let c be a shared-memory cell, and suppose that index u counsels c at time t on input I. Then by Definition 3.5 either of the following two cases applies.

Case 2.1. No processor writes into c at time t on input I and either of the following two cases applies.

Case 2.1(i). u counsels c at time $t-1$ on input I, which by the induction hypothesis implies that u affects c at time $t-1$ on input I, and so u must affect c at time t on input I.

Case 2.1(ii). There exists a processor p which writes into c at time t on input $I(u)$. On input $I(u)$, cell c contains at time t, a value which is divisible by 2^t. This is not the case on input I. Therefore u affects c at time t on input I.

Case 2.2. Some processor p writes into c at time t on input I and either of the following two cases applies.

Case 2.2(i). No processor writes into c at time t on input $I(u)$. The argument in this case is similar to Case 2.1(ii) above.

Case 2.2(ii). Some processor p' writes into cell c at time t on input $I(u)$ and either of the following two cases applies.

Case 2.2(ii)(a). $p' \neq p$. Without loss of generality, assume that $p > p'$. Then the value in cell c at time t on input I is divisible by 3^p, whereas the value in cell c at time t on input $I(u)$ is not.

Case 2.2(ii)(b). $p' = p$ and u counsels processor p at time t on input I. By Case 1 above, u affects processor p at time t on input I, and hence the values written into c at time t are different on inputs I and $I(u)$.

In either case, index u affects cell c at time t on index I.

Finally, property (iii) of Definition 3.7 is easy to verify, since the state of all processors is uniquely "stamped" with its identity number and the time. This completes the proof that M' is predictable. □

4. The lower bound.

LEMMA 4.1. *Let M be a* PRAM. *Suppose $p_u \neq p_v$ are two processors, c is a shared-memory cell, and I an input. If p_u writes into cell c at time t on input $I(u)$ and p_v writes into cell c at time t on input $I(v)$, then either u affects p_v at time t on input $I(v)$ or v affects p_u at time t on input $I(u)$.*

Proof. For a contradiction, assume the opposite and consider what happens at time t. On input $I(u)$, p_u writes into c. Since by hypothesis v does not affect p_u at time t on input $I(u)$, p_u must write into c at time t on input $I(u)(v)$. A similar argument shows that p_v must also write into c at time t on input $I(u)(v)$, which contradicts the fact that our PRAMs are exclusive-write. □

DEFINITION 4.2. (i) $K(p, t, I)$ is the set of indices which affect processor p at time t on input I.

(ii) $K(t) = \max_{p,I} |K(p, t, I)|$.

(iii) $L(c, t, I)$ is the set of indices which affect cell c at time t on input I.

(iv) $L(t) = \max_{p,I} |L(p, t, I)|$.

LEMMA 4.3. *Let M be a predictable* PRAM, *c a shared-memory cell of M, and I an input. Suppose $u \in L(c, t, I)$. For all $v \in L(c, t, I)$, $u \neq v$, there exists a processor p such that one of the following holds. Either:*

(i) *$u \in L(c, t, I(v))$, or*

(ii) *$v \in L(c, t, I(u))$, or*

(iii) *$u, v \in K(p, t, I)$.*

Proof. Suppose that hypotheses (i) and (ii) do not hold. Let $t_u \leqq t$ be the last step before t in which some processor p_u writes into c on input $I(u)$, let $t_v \leqq t$ be the last step before t in which some processor p_v writes into c on input $I(v)$, and let $t' \leqq t$ be the last step before t in which some processor p' writes into c on input I. If any of t_u, t_v, t' are undefined, set them equal to zero. Note that since $u \neq v$, at least one of t_u, t_v, t' is nonzero. There are two cases to consider.

Case 1. $t' \geqq t_u, t_v$. Since u and v both affect c at time t on input I, they must both affect p' at time t' on input I, that is, $u, v \in K(p', t', I)$, which, since M is persistent, satisfies hypothesis (iii).

Case 2. $t' < \max(t_u, t_v)$. Without loss of generality, assume that $t_u \geqq t_v, t'$. Let $S(c, t, I)$ denote the contents of cell c at time t on input I. Then

$$S(c, t_u, I(u)) = S(c, t, I(u))$$

$$= S(c, t, I(u)(v))$$

since by hypothesis v does not affect c at time t on input $I(u)$. Also, by a similar argument

$$S(c, t_v, I(v)) = S(c, t, I(u)(v)).$$

Therefore

$$S(c, t_u, I(u)) = S(c, t_v, I(v)).$$

Thus since M is predictable, we can conclude that $p_u = p_v$ and $t_u = t_v$. Thus u, $v \in K(p_u, t_u, I)$, which satisfies property (iii) since M is persistent. □

LEMMA 4.4. *Let $E \subseteq K(p, t, I)$, and $u \in E$. Let $Y(u)$ be the set of indices $v \in E$ such that either u affects p at time t on input $I(v)$, or v affects p at time t on input $I(u)$. Then* $|Y(u)| \geqq |E| - K(t-1)$.

Proof. Let $E \subseteq K(p, t, I)$ and $u, v \in E$. There are two cases to consider.

Case 1. $u \in K(p, t-1, I)$. It is sufficient to demonstrate that if $v \notin Y(u)$ then $v \in K(p, t-1, I)$. Assume that $v \notin Y(u)$, and for a contradiction assume that $v \notin K(p, t-1, I)$. Let $S(I)$ denote the state of processor p at time $t-1$ on input I. Then $S(I) = S(I(v))$ since, by hypothesis, v does not affect p at time $t-1$ on input I, and $S(I(v)) = S(I(u)(v))$ since M is persistent and (since $v \notin Y(u)$) u does not affect p at time t on input $I(v)$. But $S(I(u)) = S(I(u)(v))$ by a similar argument, while $S(I) \neq S(I(u))$ since $u \in K(p, t-1, I)$. Thus

$$S(I) = S(I(v)) = S(I(u)(v)) = S(I(u)) \neq S(I),$$

which is a contradiction. Therefore it must be the case that $v \in K(p, t-1, I)$.

Case 2. $u \notin K(p, t-1, I)$. If $v \in K(p, t-1, I)$ then by an argument similar to Case 1 (interchanging u and v) we see that $u \in Y(v)$, or equivalently, $v \in Y(u)$. If $v \notin K(p, t-1, I)$, then since u, $v \notin K(p, t-1, I)$ and M is predictable, it must be the case that both u and v affect some cell c at time $t-1$ on input I and p reads cell c at time t on input I. Thus u, $v \in L(c, t-1, I)$, and so by Lemma 4.3, one of the following holds. Either:

(i) $u \in L(c, t-1, I(v))$, which, since M is predictable, implies that $u \in K(p, t, I(v))$; that is, $v \in Y(u)$.

(ii) $v \in L(c, t-1, I(u))$, which by a similar argument implies that $v \in Y(u)$.

(iii) There exists processor p' such that u, $v \in K(p', t-1, I)$. Thus, although v may not be a member of $Y(u)$, there are at most $K(t-1)$ choices for such a v.

Thus we have shown that there are at most $K(t-1)$ members of $K(p, t, I)$ which are not members of $Y(u)$, which implies that $|Y(u)| \geqq |E| - K(t-1)$. □

The proof of Lemma 4.4 relies heavily on the fact that the PRAM in question is predictable. The corresponding result for ordinary PRAMs, which is implicit in Cook, Dwork, and Reischuk [1], gives $|Y(u)| \geqq |E| - K(t)$. This is the crux of our improvement in the lower bound.

LEMMA 4.5. *For a predictable* PRAM, *when* $t \geqq 1$, $L(t) \leqq 3 \times 4^t$.

Proof. We will prove that for a predictable PRAM, $K(0) = 0$, $L(0) = 1$ and for $t \geqq 0$,

$$K(t) \leqq K(t-1) + L(t-1), \tag{1}$$

$$L(t) \leqq 3(K(t-1) + L(t-1)). \tag{2}$$

Once this is established, it can easily be verified by induction that for $t \geqq 1$, $K(t) \leqq 4^{t-1}$ and $L(t) \leqq 3 \times 4^{t-1}$.

The proof is by induction on t. The hypothesis is certainly true for $t = 0$. Now suppose that the hypothesis is true at time $t-1$. If u affects processor p at time t on input I, then either u affects p at time $t-1$ on input I or u affects some shared-memory cell c at time $t-1$ on input I (which p subsequently reads). Therefore

$$\begin{aligned} K(p, t, I) &\leqq K(p, t-1, I) + L(c, t-1, I) \\ &\leqq K(t-1) + L(t-1), \end{aligned}$$

which implies inequality (1) as required.

Suppose that index u affects a cell c at time t on input I. Then either of the following two cases applies.

Case 1. No processor writes into c at time t on input I. Let $Y(c, t, I)$ be the set of indices u which cause some processor to write into c at time t on input $I(u)$. Then clearly

$$|L(c, t, I)| \leqq |L(c, t-1, I)| + |Y(c, t, I)|,$$

which implies that

$$|L(c, t, I)| \leqq L(t-1) + |Y(c, t, I)|. \tag{3}$$

It remains to show a bound on $|Y(c, t, I)|$. For each $u \in Y(c, t, I)$, let p_u be the processor which writes into c at time t on input $I(u)$. Let $Z(c, t, I)$ be the set of ordered pairs (u, v) such that $u, v \in Y(c, t, I)$ and either u affects p_v at time t on input $I(v)$ or v affects p_u at time t on input $I(u)$. For each u there are at most $K(t)$ choices of v which can affect p_u at time t on input $I(u)$. Thus

$$|Z(c, t, I)| \leqq 2|Y(c, t, I)|K(t)$$

(the factor of two comes from the fact that each $(u, v) \in Z(c, t, I)$ has also been counted as (v, u)).

Let $E(u)$ be the set of indices $v \in Y(c, t, I)$ which cause p_u to write into c at time t on input $I(v)$. For every $u \in Y(c, t, I)$, $(u, v) \in Z(c, t, I)$ for all $v \notin E(u)$ by Lemma 4.1. In addition, there are at least $|E(u)| - K(t-1)$ choices of $v \in E(u)$ such that $(u, v) \in Z(c, t, I)$ by Lemma 4.4. Therefore

$$\begin{aligned}|Z(c, t, I)| &\geqq |Y(c, t, I)|((|Y(c, t, I)| - |E(u)|) + (|E(u)| - K(t-1)))\\ &= |Y(c, t, I)|(|Y(c, t, I)| - K(t-1)).\end{aligned}$$

Therefore we have

$$|Y(c, t, I)|(|Y(c, t, I)| - K(t-1)) \leqq |Z(c, t, I)| \leqq 2|Y(c, t, I)|K(t),$$

which implies that

$$|Y(c, t, I)| \leqq 2K(t) + K(t-1),$$

which when substituted into inequality (3) tells us that

$$|L(c, t, I)| \leqq L(t-1) + 2K(t) + K(t-1),$$

from which we can easily deduce inequality (2) by application of inequality (1).

Case 2. Some processor p writes into c at time t on input I and either of the following two cases applies.

Case 2(i). No processor writes into cell c at time t on input $I(u)$. Then u must affect processor p at time t on input I, that is, $u \in K(p, t, I)$.

Case 2(ii). Some processor p' writes into c at time t on input $I(u)$ and either of the following two cases applies.

Case 2(ii)(a). $p' \neq p$. Then u must affect processor p at time t on input I, and again $u \in K(p, t, I)$.

Case 2(ii)(b). $p' = p$ and u affects processor p at time t on input I. Then immediately $u \in K(p, t, I)$.

In both Cases 2(i) and 2(ii) we see that

$$|L(c, t, I)| \leqq |K(p, t, I)| \leqq K(t) \leqq K(t-1) + L(t-1)$$

(making use of inequality (1)), which implies that

$$L(t) \leqq K(t-1)+L(t-1) \leqq 3(K(t-1)+L(t-1))$$

as required. □

DEFINITION 4.6. Let $B=\{0,1\}$. If $f: B^* \to B$, $I \in B^n$ is called a *critical input* for f if for all $0 \leqq u < n$, $f(I) \neq f(I(u))$. We call f a *critical function* if for all $n \geqq 1$ there is an input in B^n which is critical for f.

THEOREM 4.7. *Any* PRAM *which computes a critical function on n inputs must take time at least* $0.5 \log n - O(1)$.

Proof. Suppose M is a PRAM which computes a critical function f in time $T(n)$. Without loss of generality we can assume that M is predictable (by Lemma 3.8). Let I be a critical input for f. Then $|L(0, T(n), I)| = n$. But Lemma 4.5 tells us that $|L(0, T(n), I)| \leqq 3 \times 4^{T(n)}$. Therefore $T(n) \geqq 0.5 \log n - O(1)$. □

5. The upper bound. In this section we will demonstrate a critical function which can be computed quickly on a PRAM. For the purposes of exposition we will present an algorithm which attempts to compute the Boolean OR of its inputs. The algorithm will fail for two reasons. First, it will get the result wrong for many inputs. Second, some input symbols will be lost, that is, there will be indices u such that u does not affect the output cell when the algorithm has terminated on any input. However, we will demonstrate that the function computed by the algorithm is critical (the all-zero input will be critical), and that it is a function of sufficiently many input symbols for the required time bound to hold.

The faulty algorithm for computing the OR of n bits proceeds as follows. Suppose that n is a power of four. During each step of the algorithm, the PRAM will attempt to reduce the number of bits to be processed by a factor of four by ORing together groups of four bits. At time t there will be a set of cells $C(t)$ which contain subresults. $C(0)=\{0, 1, \cdots, n-1\}$. For each cell c at time t there will be a set of processors $P(c, t)$ which write into c at time t on some input (although at most one member of $P(c, t)$ will do so on any particular input). $P(c, 0)=\varnothing$ for $c \geqq 0$. We will ensure that $P(c_1, t) \cap P(c_2, t)=\varnothing$ when $c_1 \neq c_2$. At time t, each cell c will contain a value $v(c, t)$ which is either zero or of the form $2^{t'}$ where $t' \leqq t$ is the last time that c was written into (t' is zero if c has not yet been written into).

In the first step of the algorithm, processor p reads the contents of cell p and writes two back there if the value read was one, in parallel for $0 \leqq p < n$. Thus $C(1)=C(0)$ and $P(c, 1)=\{c\}$ for $0 \leqq c < n$. At time t, for $t > 1$, the values from cells

$$c_i = a4^{t-1} + i4^{t-2}$$

for $0 \leqq i < 4$ are examined, and an attempt is made to place a nonzero value into c_0 if at least one of their values is nonzero, in parallel for $0 \leqq a < n/4^{t-1}$. Thus by induction on t, for $t \geqq 1$ it can be shown that

$$C(t)=\{a4^{t-1} \mid 0 \leqq a < n/4^{t-1}\}.$$

At time t the following algorithm is used. Note that we adopt the convention that $\log_2 0 = 0$.

ALGORITHM 1.
Processors in $P(c_1, t-1)$ each perform the following:
  Read a value v from cell c_3
  $t_3 := \log_2 v$
  if I wrote into cell c_1 at time $t-1$ **then**
    if $t_3 \neq t-1$ **then** write 2^t into cell c_0

Processors in $P(c_2, t-1)$ each perform the following:
  Read a value v from cell c_1
  $t_1 := \log_2 v$
  if I wrote into cell c_2 at time $t-1$ **then**
    if $t_1 \neq t-1$ **then** write 2^t into cell c_0
Processors in $P(c_3, t-1)$ each perform the following:
  Read a value v from cell c_2
  $t_2 := \log_2 v$
  if I wrote into cell c_3 at time $t-1$ **then**
    if $t_2 \neq t-1$ **then** write 2^t into cell c_0

We claim that there are no write conflicts at time t. The proof is by induction on t. The hypothesis is certainly true for $t=1$. Now suppose that the hypothesis is true at time $t-1$. Thus we know that at most one processor from each of $P(c_i, t-1)$ wrote into cell c_i, respectively, at time $t-1$, for $1 \leqq i \leqq 3$. From Algorithm 1 we deduce that $P(c_0, t) = \bigcup_{i=1}^{3} P(c_i, t-1)$. Therefore the only possible write-conflicts may occur between some processors $p_1 \in P(c_1, t-1)$, $p_2 \in P(c_2, t-1)$, and $p_3 \in P(c_3, t-1)$.

Case 1. p_1 writes into cell c_1 at time $t-1$ and p_2 writes into cell c_2 at time $t-1$. Then only p_1 writes into c_0 at time t.

Case 2. p_1 writes into cell c_1 at time $t-1$ and p_3 writes into cell c_3 at time $t-1$. Then only p_3 writes into c_0 at time t.

Case 3. p_2 writes into cell c_2 at time $t-1$ and p_3 writes into cell c_3 at time $t-1$. Then only p_2 writes into c_0 at time t.

Case 4. p_1 writes into cell c_1 at time $t-1$, p_2 writes into cell c_2 at time $t-1$ and p_3 writes into cell c_3 at time $t-1$. Then no processor writes into cell c_0 at time t.

The aim of Algorithm 1 is to make c_0 at time t contain the Boolean OR of c_0, c_1, c_2, and c_3 at time $t-1$, in the sense that the former is to be nonzero if any of the latter are. This does not happen in Case 4, preventing the algorithm from computing the Boolean OR of n inputs. This is not a major concern, however, since our aim is to have the algorithm compute some critical function. Unfortunately, it does not compute a critical function of all n inputs. Let $F_1(t)$ be the number of indices which affect a cell $c \in C(t)$ at time t on the all-zero input (due to the symmetry of the algorithm, this value will be the same for all such c). Then cell $c_0 \in C(t)$ contains a nonzero value at time t if it contained a nonzero value at time $t-1$ or some processor wrote into cells c_1, c_2, or c_3 at time $t-1$. Cell c_1 is affected by $F_1(t-1)$ indices at time $t-1$, and is affected by $F_1(t-2)$ indices at time $t-2$. Therefore any processor which writes into it at time $t-1$ is affected by $F_1(t-1) - F_2(t-1)$ indices. This processor also writes into c_0 at time t. A similar argument holds for cells c_2 and c_3. Because the first step of the algorithm is a special case, we are justified in taking $F_1(0)=0$ and $F_1(1)=1$. For $t>1$,

$$F_1(t) = 4F_1(t-1) - 3F_1(t-2).$$

Therefore $F_1(t) = (3^t - 1)/2$. Thus we compute a critical function on n inputs in $\log_3 n + O(1)$ steps, a bound which appears in Cook, Dwork, and Reischuk [1].

Now suppose that the value in cells c_1, c_2, and c_3 at time $t-2$ is identical for $t>1$. This can be achieved by making the input symbols which affect c_2 and c_3 at time $t-2$ copies of the input symbols which affect c_1 at time $t-1$. We modify Algorithm

1 as follows:

ALGORITHM 2.
Processors in $P(c_1, t-1)$ each perform the following:
  Read a value v from cell c_3
  $t_3 := \log_2 v$
  if I wrote into cell c_1 at time $t-1$ **then**
    if $t_3 = 0$ **then** write 2^t into cell c_0
  else if $0 < t_3 \leqq t-2$ **then**
    if I am the smallest-numbered processor in $P(c_1, t-1)$
      then write 2^t into cell c_0
Processors in $P(c_2, t-1)$ each perform the following:
  Read a value v from cell c_1
  $t_1 := \log_2 v$
  if I wrote into cell c_2 at time $t-1$ **then**
    if $t_1 = 0$ **then** write 2^t into cell c_0
Processors in $P(c_3, t-1)$ each perform the following:
  Read a value v from cell c_2
  $t_2 := \log_2 v$
  if I wrote into cell c_3 at time $t-1$ **then**
    if $t_2 = 0$ **then** write 2^t into cell c_0

We claim that there are no write conflicts at time t. The proof is by induction on t. The hypothesis is certainly true for $t=1$. Now suppose that the hypothesis is true at time $t-1$. Let

$$v = v(c_1, t-2) = v(c_2, t-2) = v(c_3, t-2).$$

Case 1. $v > 0$. Since the values in each cell are monotonically nondecreasing with time, we know that $t_i > 0$ for $1 \leqq i \leqq 3$. Since $t_3 > 0$, the smallest numbered processor in $P(c_1, t-1)$ may write into c_0 at time t, and if any other processor from $P(c_1, t-1)$ wrote into c_1 at time $t-1$, then it is prevented from writing into c_0 at time t. Since $t_1 > 0$, no processor in $P(c_2, t-1)$ writes into c_0 at time t, and since $t_2 > 0$, no processor in $P(c_3, t-1)$ writes into c_0 at time t.

Case 2. $v = 0$. For $1 \leqq i \leqq 3$, either $t_i = 0$ or $t_i = t-1$. Therefore Algorithm 2 behaves in a manner identical to Algorithm 1, and so the same arguments prevent a write-conflict from occurring in this case.

Note that Algorithm 2 does not compute the same function as Algorithm 1. Let $F_2(t)$ be the number of indices which affect a cell $c \in C(t)$ at time t on the all-zero input (due to the symmetry of the algorithm, this value will be the same for all such c). Then $F_2(0)=0$, $F_2(1)=1$, and for $t>1$

$$F_2(t) = 4F_2(t-1) - 2F_2(t-2).$$

Therefore $F_2(t) = a((2+\sqrt{2})^t - b^{-t})$ where $a = (3\sqrt{2}+4)/4(2\sqrt{2}+3)$ and $b = 1 + (1/\sqrt{2})$.

THEOREM 5.1. *There is a critical function which can be computed on $F_2(t)$ inputs in $t+O(1)$ steps by a* PRAM *using 4^t processors.*

Proof. Suppose we are given $F_2(t)$ input bits for some $t \geqq 0$. These bits are expanded to give 4^t bits by making multiple copies of each bit according to the requirements of Algorithm 2. Each processor p, $0 \leqq p < 4^t$, examines the tree structure of the algorithm and determines which input bit that cell p should be a copy of. It then reads that bit and writes it into cell p. This takes two PRAM steps. Algorithm 2 is then executed in parallel t times. The total run time is thus $t+O(1)$ steps. □

COROLLARY 5.2. *There is a critical function which can be computed in time* $\log_c n + O(1)$ *on* n^d *processors where* $c = 2+\sqrt{2}$ *and* $d = 2/\log c < 1.13$.

6. Conclusion and open problems. We have moved the upper and lower bounds for the computation of critical functions on PRAMs closer together. The major remaining open problem is to make them meet. Our lower bound holds for PRAMs which have powerful processors and can read and write large values. Can a better lower bound be found for PRAMs which write only zeros and ones? Our upper bound differs from that of Cook, Dwork, and Reischuk [1] in that it makes use of large values. Is this necessary? Is it possible to obtain better bounds for the computation of Boolean OR (which appears to be the most interesting critical function)? The best known upper bound (which appears in [1]) is approximately $0.73 \log n + O(1)$.

REFERENCES

[1] 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.

[2] ———, *Bounds on the time for parallel* RAMs *to compute simple functions*, in Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 1982, Association for Computing Machinery, New York, 1982, pp. 231-233.

[3] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 114-118.

[4] L. M. GOLDSCHLAGER, *Synchronous parallel computation*, Ph.D. thesis, Tech. Report TR-114, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, December 1977.

[5] ———, *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073-1086.

[6] I. PARBERRY, *A complexity theory of parallel computation*, Ph.D. thesis, Department of Computer Science, University of Warwick, Coventry, England, May 1984.

[7] ———, *Parallel speedup of sequential machines: a defense of the parallel computation thesis*, SIGACT News, 18 (1986), pp. 54-67.

[8] ———, *Parallel Complexity Theory*, Research Notes in Theoretical Computer Science, Pitman, London, 1987.

[9] I. PARBERRY AND P. Y. YAN, *Improved upper and lower time bounds for parallel random access machines without simultaneous writes*, in Proc. 1989 International Conference on Parallel Processing, Vol. 3, St. Charles, IL, Penn State Press, State College, PA, August 1989, pp. 226-233.

[10] R. REISCHUK, *A lower time-bound for parallel random-access machines without simultaneous writes*, Research Report RJ3431, IBM Research Laboratories, San Jose, CA, March 1982.

[11] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, sorting and merging in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88-102.

[12] H. SIMON, *A tight* $\Omega(\log \log n)$*-bound on the time for parallel* RAM'*s to compute nondegenerated Boolean functions*, Inform. and Control, 55 (1982), pp. 102-107.

[13] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688-708.

[14] P. Y. YAN, *Lower bound techniques in some parallel models of computation*, Ph.D. thesis, Department of Mathematics, Pennsylvania State University, University Park, PA, 1989.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 100-125, February 1991
© 1991 Society for Industrial and Applied Mathematics
006

HIGH-PROBABILITY PARALLEL TRANSITIVE-CLOSURE ALGORITHMS*

JEFFREY D. ULLMAN† AND MIHALIS YANNAKAKIS‡

Abstract. There is a straightforward algorithm for computing the transitive-closure of an n-node graph in $O(\log^2 n)$ time on an EREW-PRAM, using $n^3/\log n$ processors, or indeed with $M(n)/\log n$ processors if serial matrix multiplication in $M(n)$ time can be done. This algorithm is within a log factor of optimal in *work* (processor-time product), for solving the all-pairs transitive-closure problem for dense graphs. However, this algorithm is far from optimal when either (a) the graph is sparse, or (b) we want to solve the single-source transitive-closure problem. It would be ideal to have an $\mathcal{NC}$ algorithm for transitive-closure that took about e processors for the single-source problem on a graph with n nodes and $e \geqq n$ arcs, or about en processors for the all-pairs problem on the same graph. While an algorithm that good cannot be offered, algorithms with the following performance can be offered. (1) For single-source, $\tilde{O}(n^{\varepsilon})$ time with $\tilde{O}(en^{1-2\varepsilon})$ processors, provided $e \geqq n^{2-3\varepsilon}$, and (2) for all-pairs, $\tilde{O}(n^{\varepsilon})$ time and $\tilde{O}(en^{1-\varepsilon})$ processors, provided $e \geqq n^{2-2\varepsilon}$.[1] Each of these claims assumes that $0 < \varepsilon \leqq \frac{1}{2}$. Importantly, the algorithms are (only) *high-probability* algorithms; that is, if they find a path, then a path exists, but they may fail to find a path that exists with probability at most $2^{-\alpha c}$, where α is some positive constant, and c is a multiplier for the time taken by the algorithm. However, it is shown that incorrect results can be detected, thus putting the algorithm in the "Las Vegas" class. Finally, it is shown how to do "breadth-first-search" with the same performance as can be achieved for single-source transitive closure.

Key words. transitive closure, reachability, breadth-first search, parallel algorithms, probabilistic algorithms

AMS(MOS) subject classifications. 68Q20, 68Q25, 68R10

1. Introduction. As mentioned in the abstract, we address the apparently difficult problem of doing parallel transitive-closure when the (directed) graph is sparse and/or, only single-source information is desired. We want to use less-than-linear time, and use *work*, the product of time and number of processors, that approximates the time of the best serial algorithm. An algorithm whose work is of the same order as the best known serial time is referred to as *optimal.*

For the single-source transitive-closure problem, depth-first search (see, e.g., Aho, Hopcroft, and Ullman [1974]) takes $O(e)$ time on a graph of e arcs.[2] Thus, $O(e)$ work is our target for the single-source problem. When the graph is sparse,[3] then the all-pairs transitive-closure problem can be solved by performing a depth-first search from each node, taking $O(ne)$ time; that is our target for the all-pairs problem. As seen from the abstract, we do not reach either target, except for the all-pairs case when e is fairly large. However, we make significant progress, in the sense that we have the first

* Received by the editors July 12, 1989; accepted for publication (in revised form) April 17, 1990.

† Department of Computer Science, Stanford University, Stanford, California 94305. The work of this author was partially supported by Office of Naval Research contract N00014-88-K-0166 and by a Guggenheim Fellowship.

‡ AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

[1] $\tilde{O}$ is the notation, proposed by Luks and furthered by Blum for "within some number of log factors of." That is, we say $f(n)$ is $\tilde{O}(g(n))$ if for some constants c and k, and all sufficiently large n, we have $f(n) \leqq c(\log n)^k g(n)$. Intuitively, just as "big-oh" elides constant factors, $\tilde{O}$ elides logarithmic factors.

[2] Throughout, we assume that n is the number of nodes, e the number of arcs, and that $n \leqq e$.

[3] "Sparse" normally means that $e \ll n^2$, although if we believe that one in the sequence of "fast" matrix multiplication algorithms that have been proposed (see, e.g., Coppersmith and Winograd [1987]), each taking $M(n)$ time where $M(n)$ is some power of n between two and three, will turn out to be practical, then "sparse" should be taken to mean $e \ll (M(n) \log n)/n$.

algorithms that simultaneously use time much less than linear and use work that is less than $M(n)$.

High-probability algorithms. Unless otherwise stated, all algorithms described in this paper are *high-probability* algorithms, meaning that

(1) If they report a path from one node to another, then there truly is such a path.

(2) If such a path exists, then the probability that the algorithm fails to report its existence is at most some $p_0 < 1$. By doubling the time taken by the algorithm, we can square the probability of an error. Thus, a linear-factor increase in the running time of the algorithm yields an exponential decrease in the probability of error.

In this paper, we shall describe versions of algorithms in which we only claim that there is a positive probability of finding a path when one exists. Naturally, if the algorithm is repeated $O(\log(1/\varepsilon))$ times, we can reduce the probability of failing to find a path to whatever $\varepsilon > 0$ we desire.

Moreover, we shall show that it is possible to check our result for validity with little additional work. Thus, the algorithms can be run in a "Las Vegas" mode, where termination only occurs when the correct answer has been obtained. In that case, the running times we quote should be interpreted as expected times for termination, and the probability of the actual time being greater than that by a factor c decreases exponentially with c.

$\tilde{O}$ and $\tilde{\Omega}$ notation. The algorithms we discuss introduce factors of $\log n$ from several sources. To avoid cluttering the expressions we use, these factors are elided. All our claims involve factors of at least n^{ε} for some $\varepsilon > 0$, so logarithmic factors do not dominate. As mentioned, we use the notation $\tilde{O}$ to subsume factors of $\log n$, just as the conventional "big-oh" subsumes constant factors. Similarly, when dealing with lower bounds, we use $\tilde{\Omega}$ as a version of "big-omega" that subsumes factors of $\log n$.

Previous work. Solutions for several related problems are known. Transitive-closure on an undirected graph is really the "connected components" problem. Shiloach and Vishkin [1982] gives the basic algorithm, while a series of improvements culminating in Cole and Vishkin [1986] improved the total work or elapsed time for very sparse graphs. Gazit [1986] gives an optimal randomized algorithm for the same problem.

Gazit and Miller [1988] offer an $\mathcal{NC}$ algorithm for "breadth-first search," which is really computing the distance, measured in number of arcs, from a given node. That problem generalizes single-source transitive closure, but they do not get below the $M(n)$-work barrier.

Several algorithms for transitive-closure on "random" graphs, that is, on the population of graphs constructed by picking each arc with a fixed probability, have been given (see Bloniarz, Fischer, and Meyer [1976], Schnorr [1978], and Simon [1986]). These offer expected time close to n^2, but their worst-case performance is as bad as can be—$O(n^3)$ or $O(ne)$, as appropriate. It should be emphasized that our performance measure is independent of the actual graph to which it is applied.

We should also note the paper by Broder et al. [1989], which deals with connectivity in undirected graphs. They, like us, use a technique of guessing a subset of the nodes and doing a search from each, hoping to connect all the guessed nodes that are found along a given path. Their search technique is random walks, while we use exhaustive, deterministic exploration for a limited distance. It is easy to show that random walks will not serve when directed graphs are concerned, because it is easy to intuitively "trap" a random walk on a directed graph.

Organization of the paper. In § 2 we introduce our algorithm for solving the single-source problem in $\tilde{O}(\sqrt{n})$ time and $\tilde{O}(e)$ processors. This algorithm contains

most of the ideas needed for the general case, but their use in the general case is much more complex. Section 3 provides the analysis of this algorithm. Then, in § 4 we introduce the general problem of solving transitive-closure for a graph of n nodes and e arcs, with s sources and paths of distance up to d allowed. Four reduction strategies that work with high probability are proposed.

In § 5 we give a general algorithm for the case of sparse graphs. Section 6 shows that this algorithm is the best that can be achieved using the four reductions of § 4. Section 7 gives some simplifications of the general algorithm of § 5 for interesting special cases: dense graphs, single-source, and all-pairs problems. Finally, in § 8, we show how to extend the ideas to solve the breadth-first-search problem for a single source in the same time as we can do single-source reachability.

2. A simple algorithm. Exploration of a directed graph from a single source node appears to be an inherently sequential process, especially in a case where the graph is something like a single line emanating from the source. If we are to explore a path of length $\Omega(n)$ in less than time n, we must explore from many of the nodes along the path before we even know that they are on the path. However, carried to extremes, we search from every node in parallel, and we arrive at the standard path-doubling method of solving the all-pairs problem in $\mathcal{NC}$ by doing $\log n$ Boolean matrix multiplications.

What we would like to do is to search from some of the nodes on the path forward for a short distance, until the searches link up; that is, we explore from each of the selected nodes at least as far as the next selected node, as suggested by Fig. 2.1. The searches can be done in parallel, and if nodes on the path are not too far apart, we can discover any path from the source to any node without computing the entire all-pairs transitive-closure. We need to compute the nodes reached from only a subset of the nodes, and we need to know only about the nodes reached from this subset along paths of limited length.

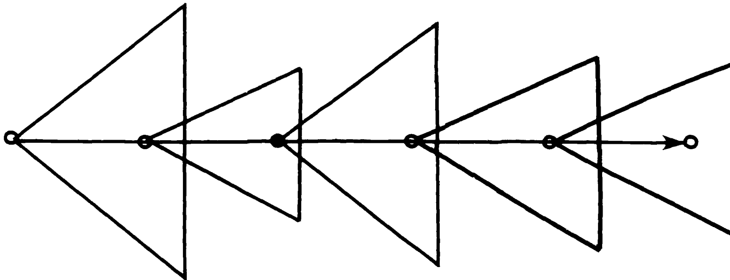


FIG. 2.1. *Finding a path by short searches.*

Unfortunately, it appears that any deterministic selection of a subset of the nodes is of little help. Given a selection of, say, half the nodes, we can always pick a graph in which the only path from the source to some node has a gap of length $n/2$ along which no selected node occurs. Thus, we would have to search distance $n/2$ from our $n/2$ selected nodes, which is almost the same as computing the full transitive-closure. However, if we pick s "distinguished" nodes at random, it is well known (see, e.g., Greene and Knuth [1982]) that a path is unlikely to have a gap longer than $O((n \log n)/s)$ with no distinguished node. More formally, we have Lemma 2.2.

LEMMA 2.2. *If we choose s "distinguished" nodes at random from an n-node graph, then the probability that a given (acyclic) path has a sequence of more than $(cn \log n)/s$ nodes, none of which are distinguished, is, for sufficiently large n, bounded above by $2^{-\alpha c}$ for some positive α.* □

The case we shall exploit in this section uses about $\sqrt{n} \log n$ distinguished nodes, and therefore needs to search forward for about $\sqrt{n}$ distance from each distinguished node. As we shall see, there is an important "trick" that works best when the number of distinguished nodes and the distance explored are as close as possible. We begin by giving the algorithm for bounded-degree graphs, and then show how it extends naturally to all graphs.

An algorithm for bounded-degree graphs. Let us now assume that all nodes have in-degree at most two and out-degree at most two. The time we shall take is $\tilde{O}(\sqrt{n})$, and we shall use n processors. The algorithm is outlined below.

ALGORITHM 2.3. Parallel, single-source transitive closure with high probability, $\tilde{O}(\sqrt{n})$ time, and n processors.

INPUT: A directed graph G with n nodes, in- and out-degree two. Also, a source node v_0.

OUTPUT: For each node of G, a decision whether the node is reached from v_0. The decision is correct with high probability, in the sense defined in § 1.

METHOD: Perform each of the following steps.

(1) Select $\sqrt{n} \log n$ nodes to be distinguished, at random. In what follows, we shall include the source v_0 among the distinguished nodes, even if it was not picked.
(2) Search from all the distinguished nodes, to find, for each node v, a set of distinguished nodes that reach v. The set for node v must include all distinguished nodes that reach v along a path of length $\sqrt{n}$ or less, and it may include other distinguished nodes reaching v along longer paths, but may not include a node that does not reach v.
(3) Construct a new graph H whose nodes are the distinguished nodes of G (including v_0). There is an arc $u \to v$ in H if it was determined in step (2) that u can reach v.
(4) Compute the all-pairs transitive closure of H, and thus determine which of the distinguished nodes are reachable from the source v_0.
(5) For each node v, determine whether there is a distinguished node w that
    (a) Reaches v along a short path, as determined in step (2), and
    (b) Is reached by v_0, as determined in step (4).

The algorithm above will detect all reachable nodes with some probability greater than zero. To reduce the error rate as far as we like (but not to zero), we can repeat steps (1)–(5) as many times as we like. For each desired error rate, there is some number of repetitions necessary, but this number does not depend on n. Combine the repetitions by saying a node v is reached from v_0 if any iteration says v is reachable.

Details of step (2). All but step (2) of Algorithm 2.3 can be accomplished in time $\tilde{O}(\sqrt{n})$ with n processors on a PRAM by straightforward means that will be reviewed in the next section. Step 2 is not hard to accomplish within these limits either. Suppose we divide the n processors equally among the distinguished nodes. Then each would get $\tilde{\Omega}(\sqrt{n})$ processors. The processors assigned to w could perform a breadth-first search from w cooperatively. If at any level, there were at most $\tilde{O}(\sqrt{n})$ new nodes, the construction of the next level could be done in $O(1)$ time, since we assume out-degree at most two. If there are more than $\tilde{O}(\sqrt{n})$ nodes at a level, we must treat

them in groups of $\sqrt{n}$. However, there are only n nodes among all the levels, so the total delay due to "excess" nodes at a level is no more than $O(\sqrt{n})$.

The above approach depends on being able to assign processors consistently, even though a node may be reached from two different predecessors. The coordination can be done, but appears to require $O(\log n)$ time to do so. The method we actually use saves this logarithmic factor in running time.

To execute step (2), we assign one processor to each node. This processor creates a table in the shared PRAM memory indexed by the distinguished nodes. The entry for distinguished node w in the table for node v tells

(a) If it is known that w can reach v.

(b) For successors x and y of v, whether w is known to reach x and whether w is known to reach y.

We also create two doubly linked lists for v of

(a) Those distinguished nodes known to reach v but not known to reach successor x.

(b) Those distinguished nodes known to reach v but not known to reach successor y.

The algorithm proceeds in $\tilde{O}(\sqrt{n})$ rounds to propagate information about which nodes are reached by which distinguished nodes. Initially, each distinguished node knows that it is reached by itself, and nothing else is known. This information is passed to its predecessors, so they can properly initialize their tables. Note that the predecessors of distinguished node w know that their successor w is "reached by w," but the predecessors do *not* know that they are reached by w themselves; indeed they may not be. However, it is important that, if they turn out to be reached by w, they do not waste time telling w this fact. This avoidance of wasted messages is the reason why nodes need to know what their successors know.

In one round, the following messages are passed between nodes, and tables are updated accordingly.

(1) If node v knows it is reached by a distinguished node w, and one of its successors x does not know that it is reached by w, then v sends x a message telling it *one* new distinguished node that reaches x. Note that each successor of v gets to learn about only one new distinguished node from each predecessor, in one round.

(2) If v has learned from one predecessor z that v is reached by distinguished node w, then v tells its other predecessor, if it has one, that v now knows it is reached by w. Of course, z also knows that v now knows about w, so each predecessor of v can update its table accordingly.

If there are s distinguished nodes, and we want to propagate reachability information for distance at least d, then we require only $d+s$ rounds to do so, as we shall prove in § 3. The intuitive reason is that, while a fact can be delayed in its propagation because a given node v has many other facts to tell one of its successors, no one fact can be delayed twice by the same fact. However, the true picture is somewhat more complex than that, since it is often unclear which fact causes the delay of another fact. In the case at hand, where s and d are both $\tilde{O}(\sqrt{n})$, we require $\tilde{O}(\sqrt{n})$ rounds, each of which can be executed in $O(1)$ time on a PRAM, to accomplish step (2) of Algorithm 2.3.

The unlimited-degree case. If the graph does not have in- and out-degree two, we begin with step (1) of Algorithm 2.3, selecting distinguished nodes at random. However, before proceeding to step (2), we convert the graph G to a graph G' that does have in- and out-degree two. For each node v with more than two successors, we create a

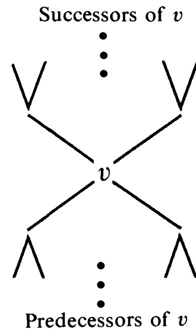


FIG. 2.4. *Forcing in- and out-degree two.*

balanced binary tree to fanout to those successors, and for each node v with more than two predecessors, we create a balanced binary tree to fanin, as suggested by Fig. 2.4.

The number of introduced nodes for a node v is no greater than the sum of the in- and out-degrees of v. Thus, G' has $O(e)$ nodes, if e is the number of edges of G. Since G' clearly has in- and out-degree two, G' also has $O(e)$ edges.

Moreover, since the trees are balanced, no path through an introduced tree is longer than $\log n$. As a result, to search for distance d in G, it suffices to search for distance $d \log n$ in G'. In particular, we can now complete steps (2)–(5) of Algorithm 2.3 on the graph G', but by searching distance $\sqrt{n} \log n$, instead of distance $\sqrt{n}$, and by using e processors instead of n. Since in step (2) we have $\sqrt{n} \log n$ distinguished nodes, and we search for the same distance, the number of rounds $(s+d)$ that we need does not even go up by more than a factor of 2.[4] Our conclusion is that it is possible to do single-source transitive-closure for an arbitrary graph in $\tilde{O}(\sqrt{n})$ time with e processors.

3. Correctness and efficiency of the simple algorithm. We now need to show that the algorithm described in § 2 is correct with high probability, and we must show that its running time and processor utilization are as claimed in that section. The key point is the correctness of step (2) of Algorithm 2.3, which is implied by the first lemma.

LEMMA 3.1. *The algorithm for step* (2) *described in the previous section, when run with s distinguished nodes, will find all distinguished nodes that reach any given node along a path of distance d or less, provided we run the algorithm for at least $s+d$ rounds.*

Proof. Consider a path $v_0 \rightarrow \cdots \rightarrow v_k$ along which propagates the information that distinguished node v_0 reaches node v_k. After r rounds, suppose that the reachability of v_0 has propagated to v_i, but not to v_{i+1}; that is, v_i knows it is reached by v_0, but v_{i+1} does not.[5] Let the *delay* be $\delta = r - i$, that is, the number of rounds on which v_0 failed to make progress along this path. We claim, and shall prove by induction on r, that there is a "wedge of knowledge," suggested in Fig. 3.2, where v_{i+1} knows about at least δ different distinguished nodes that reach it, v_{i+2} knows about at least $\delta - 1$, and so on.

Formally, we shall show by induction on r that if $v_0, \cdots, v_i$ know about v_0, and v_{i+1} does not, then for $j = 1, \cdots, \delta = r - i$, v_{i+j} knows about at least $\delta + 1 - j$ distinguished nodes that reach it. The basis, $r = 0$, is trivial, since then $i = 0$ and $\delta = 0$, so the "wedge" has height zero.

[4] But note that had we started with a bounded degree graph, we could choose $\sqrt{n \log n}$ sources and search for exactly that distance, thus improving the running time of step (2) by a factor of $\sqrt{\log n}$.

[5] Note that the reachability of v_0 may be known further along the path, because that information has propagated by another route.

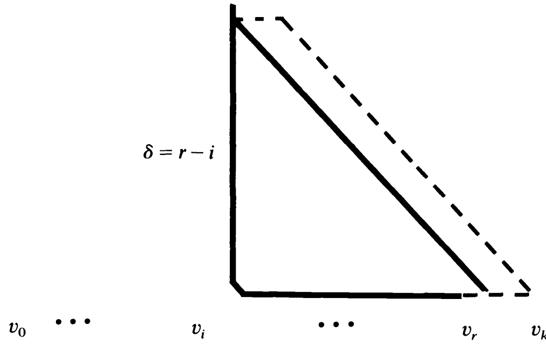


FIG. 3.2. *Wedge of knowledge.*

Now, suppose we have proceeded for r rounds, and the inductive hypothesis holds. Consider what happens on round $r+1$. First, make the following observation.

(*) If at some time, v_p knows about strictly more distinguished nodes than v_{p+1}, then on the next round v_{p+1} will learn at least one new fact.

Thus, the number of facts known by each of $v_{i+2}, \cdots, v_{r+1}$ will, after round $r+1$, be at least one greater than that implied by the solid wedge of Fig. 3.2; this profile is suggested by the dashed wedge in that figure. That is, by (*), we deduce that for $j = 2, \cdots, \delta+1$, node v_{i+j} knows about at least $\delta+2-j$ distinguished nodes that reach it by round $r+1$.

Case 1. v_{i+1} does not learn about v_0 at round $r+1$. Then v_i must tell v_{i+1} something else, because at each round, a node tells each successor something new if it can. Thus, after round $r+1$, v_{i+1} knows about at least $\delta+1$ distinguished nodes. Thus, for $j = 1, \cdots, r+1-i$, node v_{i+j} knows about at least $\delta+2-j$ distinguished nodes. The delay at round $r+1$ is $\delta+1$, so the inductive hypothesis is proved.

Case 2. v_{i+1} learns about v_0 at round $r+1$. Suppose that now $v_0, \cdots, v_m$ know about v_0, but v_{m+1} does not. If $m \geqq r+1$, the inductive hypothesis holds vacuously. Otherwise, the new delay is $\delta' = r+1-m$. Since

(1) $m > i$,

(2) $\delta' = \delta - m + i + 1$, and

(3) for $j = 2, \cdots, \delta+1$, node v_{i+j} knows about $\delta+1$ distinguished nodes,

we know that for $j' = 1, \cdots, \delta'$, node $v_{m+j'}$ knows about $\delta'+1-j'$ distinguished nodes (substitute $j'+m-i$ for j in (3)). Thus, the inductive hypothesis holds for round $r+1$.

Now, we observe that there are only s distinguished nodes that any node can ever know about. Thus, s is a limit on the height of a "wedge." That is, the delay cannot exceed s. Suppose that after r rounds, v_k does not know about v_0 yet. Then there is some $i < k$ such that $v_0, \cdots, v_i$ know about that v_0 and v_{i+1} does not. The delay is $r - i$, which is strictly greater than $r - k$. Thus, $s > r - k$.

Finally, consider $r = d + s$. We deduced that $s > r - k = d + s - k$, or equivalently, $k > d$. That is, after $s + d$ rounds, if v_k does not know about v_0, then $k > d$. Put another way, if some node v is ignorant of the fact that v_0 reaches v, then there can be no path of length d or less from v_0 to v. Thus, after $s+d$ rounds, all nodes learn about all distinguished nodes that reach them along paths of length d or less. □

Correctness of the algorithm. We now need to check that the remaining steps of Algorithm 2.3 and its extension to the general case are correct. It is easy to check that whenever a path is discovered, there is truly a path, so we need only to verify that if a path exists, it is discovered with positive probability. We may then argue that by increasing the number of distinguished nodes by a constant factor, the probability of error can be made as small as we wish.

Lemma 2.2 assures us that step (1), the selection of random distinguished nodes, leaves us a positive probability that for every node v, the shortest path from the source to v is *gapless*; that is, it has no gap of more than $\sqrt{n}$ consecutive, unselected nodes. Gaps of length $\sqrt{n}$ cannot grow to more than $\sqrt{n}\log n$ nodes after we convert graph G to the graph G' by adding fanin and fanout trees for the general case. Lemma 3.1 just showed that step (2) finds the paths between successive distinguished nodes on any gapless path. If we run for $2\sqrt{n}\log n$ rounds in step (2), then we are sure to have the necessary connections between distinguished nodes.

In particular, if our path is gapless, and w is the last distinguished node on the path (w may be the source, v_0), then steps (3) and (4) (construction of the graph H and computation of its transitive-closure) determine that there is a path from v_0 to w. Also, as the path is gapless, step (2) established that there is a path from w to v, the last node on the path. Thus, step (5) deduces that there is a path from v_0 to v. We have thus proved Theorem 3.3.

THEOREM 3.3. *The algorithm of* § 2 *determines whether there is a path from* v_0 *to each node* v, *with high probability.* □

Analysis of the algorithm. Now, we need to verify that each of the steps of Algorithm 2.3 and its extension to unbounded-degree graphs can be performed in $\tilde{O}(\sqrt{n})$ time on a PRAM with n processors. First, let us agree on a representation for graphs in the shared memory of the PRAM. The nodes will be assumed to be numbered $1, 2, \cdots, n$. Suppose that there is an array indexed by nodes, giving the number of in- and out-arcs for each node, a pointer to an array where the successors of the node are listed, and a pointer to an array where the predecessors of the node are listed.

Step (1), the selection of $\sqrt{n}\log n$ random nodes, can be done by one processor in the requisite time. However, further on in this paper, we shall need to do somewhat better: generate $\tilde{O}(n^{1-\varepsilon})$ random nodes in $\tilde{O}(n^{\varepsilon})$ time, using n processors. We can do so by the following steps.

(1) Each of the n processors selects a random node and creates a record (i, j), where j is the processor number and i the number of the node selected.

(2) The records are sorted lexicographically.

(3) Each record r looks at its predecessor to see if it has the same node (first component). If so, r is eliminated. Thus, for each selected node, only one record remains, and it is the record with the lowest-numbered processor that selected that node.

(4) Sort the remaining records by processor number, and select as many as desired from the front of this list, to be the chosen random nodes.

Thus, the entire process takes polylog time.

Step (2) of Algorithm 2.3 was analyzed in Lemma 3.1. There we proved that $\tilde{O}(\sqrt{n})$ rounds sufficed. We have only to observe that each round can be accomplished in $O(1)$ time. Message passing is done by creating for each node four words in the shared memory, for receiving messages from its predecessors and successors. Each arc into or out of the node is associated with a particular word. To send a message, a node writes into the proper word belonging to the receiving node.

For step (3), note that the graph H has $\tilde{O}(\sqrt{n})$ nodes and $\tilde{O}(n)$ arcs. Its arcs can be read from the tables created in step (2) for the distinguished nodes. That is, assign $\tilde{O}(1)$ arcs to each processor. To consider whether there is an arc $u \to v$ in H, the processor goes to the table for v and the entry for u. To get H into our form for graphs, we can compact the lists of predecessors and successors in $\tilde{O}(1)$ time by assigning $\sqrt{n}$ processors to each list and performing a parallel prefix operation. Thus, the entire construction of H is accomplished in $\tilde{O}(1)$ time.

Step (4), taking the transitive closure of H, can be done in $\tilde{O}(\sqrt{n})$ time with n processors by the ordinary, path-doubling-by-matrix-multiplication method. That is, for a graph with $\tilde{O}(\sqrt{n})$ nodes, the method usually is done in $\log^2 n$ time with $\tilde{O}(n^{3/2})$ processors, but we can trade time for processors by "Brent's theorem" (Brent [1974]), to achieve the desired bounds. Now, we look at the successors of the source v_0 in the transitive closure of H, and make a table, or vector, indicating for each distinguished node whether that node is reached in H from v_0. This part requires $O(1)$ time with n (or even $\sqrt{n}$) processors.

In step (5), we assign one processor to each node v, and examine the table of distinguished nodes that reach v. For each distinguished node, we examine the table constructed in step (4) to see whether v_0 reaches that distinguished node. The table lookup requires $O(1)$ time, so the time for considering all distinguished nodes is $\tilde{O}(\sqrt{n})$.

Finally, we must note the difference when the initial graph G does not have bounded degree. Now we are given e processors to convert G to G' by introducing fanin and fanout trees, as described at the end of § 2. First, assign to each node a number of processors equal to the out-degree of that node. The fanout tree can then be constructed in $\tilde{O}(1)$ time by obvious means. Similarly, we build the fanin trees in the same time. From that point, we proceed as in Algorithm 2.3, but with e processors, at most $n+2e$ nodes, and at most $5e$ arcs. In step (2), there are $\tilde{O}(\sqrt{n})$ distinguished nodes that must be explored for distance $\tilde{O}(\sqrt{n})$, so the time bound $\tilde{O}(\sqrt{n})$ still holds, as long as there are n processors. Steps (3) and (4) do not change, since H still has $\tilde{O}(\sqrt{n})$ nodes, and we have $e \geqq n$ processors. Step (5) is similar, since we can still assign one processor to each node and do the work in $\tilde{O}(\sqrt{n})$ time. We have thus proved Theorem 3.4.

THEOREM 3.4. *We can compute with high probability the single-source transitive-closure of an n-node, e-arc graph in* $\tilde{O}(\sqrt{n})$ *time on a* PRAM *with e processors.* □

4. High-probability reductions for transitive closure. In this section we shall consider the general "sparse" problem, which we call $S(s, d, n, e)$, of determining, in an $\tilde{O}(n)$-node, $\tilde{O}(e)$-arc graph, what nodes are reached along paths of distance at most $\tilde{O}(d)$, from each of $\tilde{O}(s)$ source nodes. We may optionally include in our answer other nodes reached from a source, but only along paths of length greater than $\tilde{O}(d)$. Our answer must be correct with high probability.

We shall assume that there is a time limit $\tilde{O}(t)$ for obtaining the answer, and we need to compute the necessary work, that is, the product of the time taken and the number of processors used. For example, the problem studied in the previous two sections is $S(1, n, n, e)$, with a time limit $t=\sqrt{n}$. The other problem of significant interest is $S(n, n, n, e)$, the all-pairs problem. Note that factors of $\log n$ are elided in all parameters, and, of course, will be elided in the measure of work.

While we are probably not very interested in instances other than the two mentioned, we need to consider this more general framework because we have devised a number of "reductions" among instances of the general problem, so a solution to a

case that interests us can involve the solution to subproblems that appear less interesting. For example, in § 2, we reduced the problem $S(1, n, n, e)$ to the problems $S(\sqrt{n}, \sqrt{n}, n, e)$ and $S(\sqrt{n}, \sqrt{n}, \sqrt{n}, n)$. The former is the problem we solved in step (2) of Algorithm 2.3, and the latter is the all-pairs transitive-closure of the graph H.

We shall also have need to consider the "dense" problem on occasion. We let $D(s, d, n)$ stand for $S(s, d, n, n^2)$.

We use two "basis" (nonrecursive) rules for solving problems and four recursive rules. Each of these rules has some role to play, although in certain circumstances we shall show that one dominates the others as far as reducing the work is concerned. Incidentally, the reader may legitimately worry if, as we elide constant factors and factors of $\log n$, whether a recursion that hides, say, a factor of $\log n$ at each of n levels, is not really hiding a dominant factor. However, when we analyze the rules for optimal strategies, we shall see that in no case do we need to recurse more than twice. Thus, hidden factors cannot accumulate.

Basis Rule B1. Our first strategy is to solve the all-pairs transitive-closure problem for the graph. We may use this strategy with any time limit whatsoever, since even $t=1$ really means polylog time. We take work $\tilde{O}(n^3)$ to solve transitive-closure this way. This strategy may seem useless, but in fact it is needed for certain subproblems with a small number of nodes, as we found for graph H in Algorithm 2.3.

Basis Rule B2. This rule generalizes the method for step (2) of Algorithm 2.3. Divide the s sources into $\lceil s/d \rceil$ groups. As in step (2) of Algorithm 2.3, use e processors to search forward from the $O(d)$ sources of each group for distance d. By Lemma 3.1, this operation requires $\tilde{O}(d)$ time. We therefore put the work for this basis rule at $es+ed$.[6] Technically, the work is $\tilde{O}(e\lceil s/d \rceil d)$, but the reader can easily check that $\tilde{O}(es+ed)$ is the same. That is, if $s \leqq d$, then $\lceil s/d \rceil = 1$, and the term ed predominates. If $s > d$, $\lceil s/d \rceil d$ is about s, and es predominates.

Recursive Rule R1. We can solve $S(s, d, n, e)$ by the following steps.

(1) Select s_1 distinguished nodes at random, from among all the nodes of the graph. We may assume with high probability that there are no gaps greater than $(n \log n)/s_1$ in paths, so search forward from each of the distinguished nodes for this distance, which is $\tilde{O}(n/s_1)$.

(2) Add to the given graph all arcs $u \to v$ between distinguished nodes that are discovered in step (1). Note that at most $\tilde{O}(s_1^2)$ arcs are added.

(3) In the resulting graph, search forward from the original s sources for distance that is sufficient to reach any node from any source. This distance is $\tilde{O}(ds_1/n + n/s_1)$.

The intuitive reason the distance suggested in (3) suffices is suggested in Fig. 4.1. A path of length d in the original graph is collapsed in the new graph so that we only have to follow from the source to the first distinguished node (at most n/s_1, since we assume no "big" gaps), then along introduced arcs to the last distinguished node, and finally to the end of the path (again, at most distance n/s_1). If there are three distinguished nodes within distance n/s_1 along the path, then we can skip the middle one. Thus, if there is distance at most d along the path, and we follow more than $2ds_1/n$ introduced arcs between distinguished nodes, we can find a shorter path, assuming there are no gaps longer than n/s_1. We conclude that distance $\tilde{O}(n/s_1)$ suffices for the *prefix* and *suffix* of the path (outside the first and last distinguished

[6] Recall that we are dropping logarithmic factors from our work estimates, as well as from all parameters. The work is really $\tilde{O}(es+ed)$.

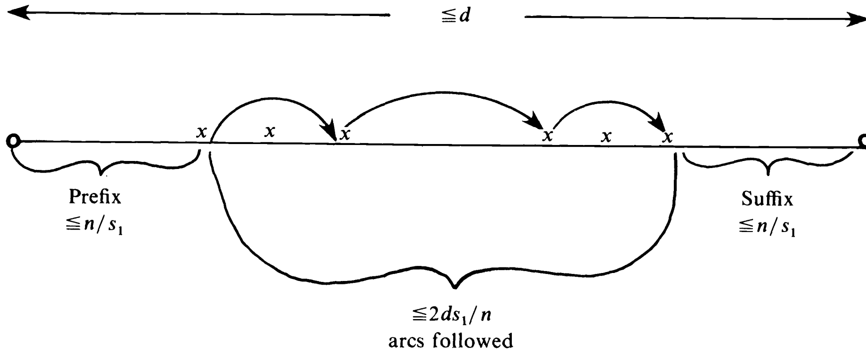


FIG. 4.1. *Shortening paths in rule* R1.

nodes), and $\tilde{O}(ds_1/n)$ suffices in the middle, jumping among distinguished nodes. In the special case in which the path has no distinguished nodes, we may assume that it has length no greater than the maximum gap length, or n/s_1.

Note that the subproblem of step (1) is $S(s_1, n/s_1, n, e)$ and the subproblem of step (3) is $S(s, ds_1/n + n/s_1, n, e + s_1^2)$. Step (2), the addition of arcs, can be done with work proportional to the number of those arcs, which could not be greater than the work of step (2), constructing those arcs in the first place. Furthermore, the addition of the arcs to the graph can be done in parallel, to the extent allowed by the number of processors. Thus, we claim that the work of step (2) could not exceed that of step (1). Consequently, we shall write rule R1 as

$$\text{(R1)} \qquad S(s, d, n, e) \to S\left(s_1, \frac{n}{s_1}, n, e\right) + S\left(s, \frac{ds_1}{n} + \frac{n}{s_1}, n, e + s_1^2\right).$$

The arrow can be interpreted as saying that the work of the problem on the left is no greater than the sum of the costs of the problems on the right. It can also be interpreted as saying that if we can solve the problems on the right with high probability, then we can solve the problem on the left with high probability, using, within some logarithmic factors, no more time or work than the sum of what is used to solve the problems on the right.

Recursive Rule R2. A similar strategy begins with step (1) of R1. However, we next consider the "dense" problem of computing the transitive-closure of the graph consisting of the distinguished nodes only, and all arcs that were discovered among them in step (1); this graph is analogous to H in Algorithm 2.3. Referring to Fig. 4.1, once we take the transitive-closure and then install the resulting arcs in the original graph, we can leap from the first distinguished node to the last, in one arc. The distance we need to follow is thus just the length of the prefix and suffix, plus one, which is $\tilde{O}(n/s_1)$, as suggested by Fig. 4.2. That is also a bound on the length of a path that is so short it has no distinguished nodes.

We may thus express R2 as

$$\text{(R2)} \qquad S(s, d, n, e) \to S\left(s_1, \frac{n}{s_1}, n, e\right) + D\left(s_1, \frac{ds_1}{n}, s_1\right) + S\left(s, \frac{n}{s_1}, n, e + s_1^2\right).$$

Recall that D denotes the dense case, where e is the square of the number of nodes, s_1^2 here. The middle term represents the transitive-closure, and the last term represents the final step, where we search for distance $\tilde{O}(n/s_1)$ from the original s sources in the augmented graph.

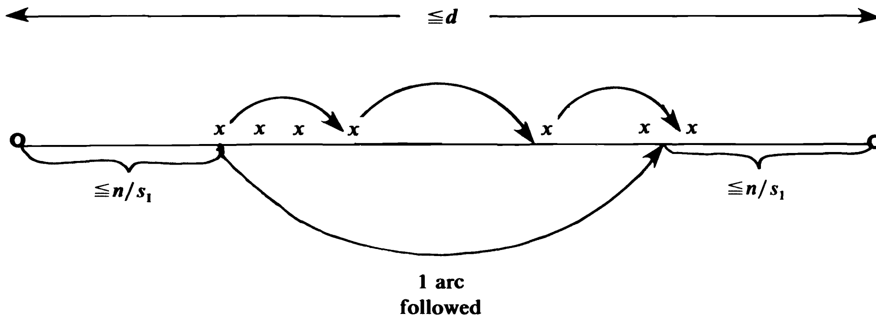


FIG. 4.2. *Shortening paths in rule* R2.

Recursive Rule R3. In this variation, we do the following.

(1) Select s_1 distinguished nodes at random and search forward from them for distance $\tilde{O}(n/s_1)$, as in R1 and R2.

(2) Reverse the arcs and search backward in the graph from the same nodes for the same distance.

(3) Construct a new graph whose nodes are the distinguished nodes and sources of the original graph, a total of at most $s+s_1$ nodes. Add to the new graph the arcs found in step (1) between distinguished nodes. Also add the arcs found in (2) from a source to a distinguished node. That is, if distinguished node w was found to reach source v following arcs backward, then add arc $v \rightarrow w$. The maximum number of arcs in the new graph is s_1^2 from step (1) and ss_1 from step (2).

(4) In the graph constructed in step (3), search forward from the sources for distance $2ds_1/n$. As we argued concerning R1, this distance is sufficient to get us from any source to the last distinguished node on any path from that source.

(5) Add to the original graph all the arcs $v \rightarrow w$ such that v is a source node and w a distinguished node found reachable from v in step (4). The number of arcs added to the original graph is at most ss_1.

(6) In the graph constructed by step (5), search forward from the sources for distance $\tilde{O}(n/s_1)$. This distance suffices to follow any path of length d in the original graph, since in the augmented graph we can go in one arc to the last distinguished node, and then need only to follow the suffix. Similarly, a path with no distinguished nodes is assumed to be no longer than n/s_1.

As for R1, we can argue that steps (3) and (5) can be performed with work that does not exceed the work of constructing the arcs, and that these steps can be performed with the maximum parallelism allowed by the number of processors. Thus we may neglect the cost of these steps. The work of constructing a new graph in step (2) with the arcs reversed is no greater than $\tilde{O}(e)$, and it can be done with the maximum amount of parallelism allowed by the number of processors, down to $\tilde{O}(1)$ time. The details require some thought, but are similar to the techniques already mentioned in the construction leading to Theorem 3.4. Since the initial subproblem surely requires work $\tilde{\Omega}(e)$, or else we cannot even look at all the arcs, we shall neglect the cost of reversing the arcs. We thus can express R3 as

$$\text{(R3)} \quad S(s, d, n, e) \rightarrow S\left(s_1, \frac{n}{s_1}, n, e\right) + S\left(s, \frac{ds_1}{n}, s+s_1, s_1(s+s_1)\right) + S\left(s, \frac{n}{s_1}, n, e+ss_1\right).$$

The first term represents the cost of the searches in steps (1) and (2); note that the multiplier 2 would be appropriate but unnecessary. The second term represents step (4), and the last term represents step (6).

Recursive Rule R4. The last rule is rather different from the first three, and also quite simple. We split the number of sources into some number of groups, say k, and treat the groups independently. We shall see that in practice, this trick simplifies things when $s > d$, but it is never formally necessary; it was, in fact, incorporated into basis rule B2. The description of R4 is

$$S(s, d, n, e) \to kS\left(\frac{s}{k}, d, n, e\right). \tag{R4}$$

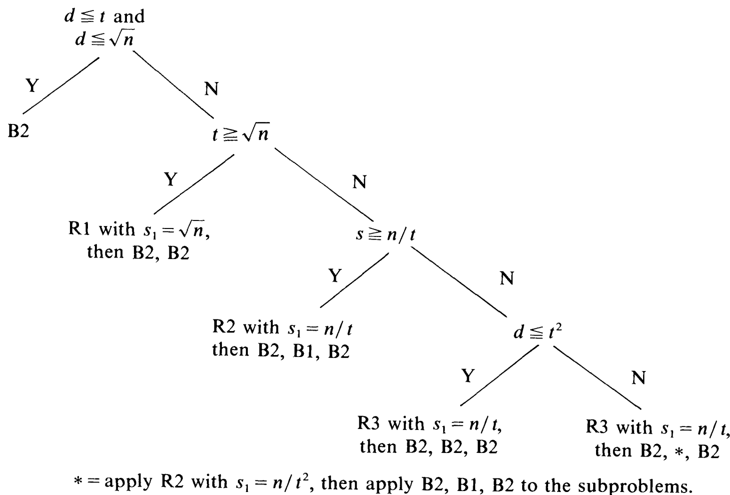
We may summarize the arguments given in connection with each of the rules by Theorem 4.3.

THEOREM 4.3. *Each of the rules* R1, R2, R3, *and* R4 *is correct, in the sense that if the subproblems on the right can be solved with high probability in a certain amount of time and work, then the problem on the left can likewise be solved with high probability, with the same amount of work, neglecting factors of* log n. □

5. A general strategy. It turns out that we can solve the problem $S(s, d, n, e)$ for all values of the parameters, and all time limits t, by a fairly simple algorithm that applies at most two recursive rules and then applies one of the basis rules to each of the resulting subproblems. The strategy is suggested by the tree of Fig. 5.1. More formally, the strategy is given by the following algorithm.

ALGORITHM 5.2. Solution to the generalized, sparse transitive-closure problem.
INPUT: A problem $S(s, d, n, e)$ and a time limit t.
OUTPUT: A strategy that solves the given problem in $\tilde{O}(t)$ time on a PRAM and uses as few processors as any strategy built from the rules B1, B2, R1, R2, R3, and R4 of the previous section.
METHOD: The algorithm consists of the following decisions.

(1) If $d \leqq t$ and $d \leqq \sqrt{n}$ then apply basis rule B2.
(2) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$), if $t \geqq \sqrt{n}$ then apply recursive rule R1 with $s_1 = \sqrt{n}$, and then apply B2 to the two subproblems that result.
(3) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$, and also $t < \sqrt{n}$), if $s \geqq n/t$ then apply R2 with $s_1 = n/t$. Apply B2 to the first and last subproblems, and apply B1 to the middle subproblem $S(s_1, ds_1/n, s_1, s_1^2)$.



$*$ = apply R2 with $s_1 = n/t^2$, then apply B2, B1, B2 to the subproblems.

FIG. 5.1. *Summary of Algorithm* 5.2.

(4) Otherwise (i.e., $d > t$ and/or $d > \sqrt{n}$, also $t < \sqrt{n}$, and also $s < n/t$), apply R3 with $s_1 = n/t$. If $d \leqq t^2$, apply B2 to each of the three generated subproblems. If $d > t^2$, there is not enough time to apply B2 to the middle subproblem, so use B2 for the first and third subproblems, and apply R2 with $s_1 = n/t^2$ to the middle subproblem. Apply B2, B1, and B2, respectively, to the three subproblems so generated.

Analysis of Algorithm 5.2. Let us now compute the function that gives the work required by Algorithm 5.2. That function has five cases, corresponding to the five leaves of Fig. 5.1. The table of Fig. 5.3 gives the definitions of the cases and the work for each. Recall that all functions elide factors of log *n*.

THEOREM 5.4. *The work used by the strategy of Algorithm* 5.2 *is as given by Fig.* 5.3.

Proof. Case 1 corresponds to the leftmost leaf of Fig. 5.1. The basis B2 is used, and Lemma 3.1 tells us that $\tilde{O}(d)$ rounds suffice. Since each round takes $O(1)$ time, and the number of processors used is $e\lceil s/d \rceil$, the work is $es + ed$, as mentioned when we introduced rule B2. Since $d \leqq t$ is given, we finish within the time limit.

Case	Condition	Work
1	$d \leqq t,\ d \leqq \sqrt{n}$	$es + ed$
2	$t \geqq \sqrt{n},\ d > \sqrt{n}$	$es + e\sqrt{n}$
3	$d > t,\ t < \sqrt{n},\ s \geqq n/t$	$es + sn^2/t^2$
4	$t < d \leqq t^2,\ t < \sqrt{n},\ s < n/t$	$en/t + sn^2/t^2$
5	$d > t^2,\ t < \sqrt{n},\ s < n/t$	$en/t + sn^2/t^2 + n^3/t^4$

FIG. 5.3. *Work used by Algorithm* 5.2.

Case 2 corresponds to the second node in Fig. 5.1, where we apply R1 with $s_1 = \sqrt{n}$. The two subgoals in R1 reduce to $S(\sqrt{n}, \sqrt{n}, n, e)$ and $S(s, d/\sqrt{n} + \sqrt{n}, n, e+n)$. Since $d \leqq n$ can be assumed, and $n \leqq e$ is a global assumption, the second term can be simplified to $S(s, \sqrt{n}, n, e)$. B2 applied to the first requires $e\sqrt{n}$ work, and B2 applied to the second requires $es + e\sqrt{n}$ work. Since $t \geqq \sqrt{n}$ is assumed in this case, we finish within the time limit.

Case 3 is for the third node of Fig. 5.1, where R2 is applied with $s_1 = n/t$. The terms of R2 simplify in this case to $S(n/t, t, n, e)$, $D(n/t, d/t, n/t)$, and $S(s, t, n, e + n^2/t^2)$. B1 applied to the middle term takes work n^3/t^3 and can be done within any time limit, since it requires only polylog time, and even $t = 1$ really means $t = \tilde{O}(1)$ according to our conventions. The first term, with basis rule B2, requires $en/t + et$, and the third term with B2 requires $(e + n^2/t^2)(s + t)$. If we expand all these terms we get

$$\frac{n^3}{t^3} + \frac{en}{t} + et + es + \frac{sn^2}{t^2} + \frac{n^2}{t}.$$

Because $e \geqq n$, we can eliminate the last term in favor of the second. As $s \geqq n/t$ is assumed in Case 3, we can eliminate the first term in favor of the fifth, and we can eliminate the second in favor of the fourth. Finally, we claim that $s > t$, because $t < \sqrt{n}$ is given. That, with $s \geqq n/t$, gives us $s > \sqrt{n}$, which in turn exceeds t. Thus, the third term, et, can be eliminated in favor of the fourth, es.

Cases 4 and 5 correspond to the last two nodes of Fig. 5.1, where $d > t$, $t < \sqrt{n}$, and $s < n/t$ are assumed, and R3 is used. R3, with $s_1 = n/t$ simplifies to the subgoals $S(n/t, t, n, e)$, $S(s, d/t, n/t, n^2/t^2)$, and $S(s, t, n, e + sn/t)$. B2 applied to the first subgoal yields work $e(n/t + t)$, but since $t < \sqrt{n}$, we know that $n/t > t$, and we can write this work as en/t. B2 applied to the last subproblem requires work $(e + sn/t)(s + t)$.

In Case 4, where $d \leqq t^2$, that is, $d/t \leqq t$, there is enough time to apply B2 to the middle subgoal. Thus, the work for that subgoal is $n^2(s+d/t)/t^2$. These expressions, expanded out, are

$$\frac{en}{t}+es+et+\frac{s^2n}{t}+sn+\frac{sn^2}{t^2}+\frac{dn^2}{t^3}. \tag{5.5}$$

Remembering that $n \leqq e$ and $s < n/t$ eliminates the second and fifth terms in favor of the first. Again reasoning that $t < n/t$ whenever $t < \sqrt{n}$ eliminates the third term in favor of the first. In Case 4, we have $d/t \leqq t$, so we can eliminate the seventh term by noting that $dn^2/t^3 \leqq n^2/t$. Then we use $n \leqq e$ to eliminate the last term in favor of the first. Finally, we use the fact that $s < n/t$ to eliminate the fourth term in favor of the sixth. We are thus left with only the first and sixth terms, $en/t + sn^2/t^2$.

In Case 5, we have $d > t^2$, so we use R2 with $s_1 = n/t^2$ on the middle subgoal. The first and last subproblems when we initially expanded $S(s, d, n, e)$ by R3 yield the first five terms of (5.5), which by the reasoning above simplifies to $en/t + s^2n/t$. The middle term $D(s, d/t, n/t)$, which stands for $S(s, d/t, n/t, n^2/t^2)$, when expanded by R2 yields the subproblems $S(n/t^2, t, n/t, n^2/t^2)$, $S(n/t^2, d/t^2, n/t^2, n^2/t^4)$, and $S(s, t, n/t, n^2/t^2)$. Note that we can neglect the "$+s_1^2$" term in the number of edges of the last subproblem, because the graph is already dense. If we apply B2 to the first, we get work $n^3/t^4 + n^2/t$; B1 applied to the second uses work n^3/t^6; and B2 on the third uses $sn^2/t^2 + n^2/t$. Thus, all the terms in the expression for work are

$$\frac{en}{t}+\frac{s^2n}{t}+\frac{n^3}{t^4}+\frac{n^2}{t}+\frac{n^3}{t^6}+\frac{sn^2}{t^2}.$$

Evidently, the fifth term can be eliminated in favor of the third, and the fourth in favor of the first. The second can be eliminated in favor of the sixth, because $s < n/t$ is assumed in Case 5. Thus, the remaining terms are $en/t + sn^2/t^2 + n^3/t^4$, as stated in Fig. 5.3. □

A Las Vegas algorithm. We can modify Algorithm 5.2 to check the validity of its answer, using $O(es)$ work and $\tilde{O}(1)$ time. As we shall see in the next section, the work required by Algorithm 5.2 is $\tilde{\Omega}(es)$ in any of the five cases of Fig. 5.3; thus the additional work and time used by this modification can be neglected. Suppose we have a set of nodes X deemed by the algorithm to be reachable from source node v. Examine the arcs out of each node in X, in parallel, and if any are found to enter a node not in X, then we conclude that our answer is incorrect. However, if for each source node, the set of reachable nodes is closed under successor, then we claim the answer is correct. We cannot say we reach a node not actually reached, because all paths found are real paths. We cannot say we fail to reach a node that we actually reach, or else some set X, ostensibly the nodes reachable from some source v, would have an arc to a node not in X.

We can thus modify Algorithm 5.2 to be a Las Vegas algorithm by adding the check for validity just described. If the check fails, then we repeat the algorithm, until eventually the test succeeds. Since there is positive probability that the algorithm succeeds on any given round, the expected time of the algorithm is still $\tilde{O}(t)$, and the expected work is as given in Fig. 5.3.

6. Optimality of the general algorithm. We shall now show that Algorithm 5.2 is the best we can do, given the tools at hand—recursive rules R1 through R4 and basis rules B1 and B2. First, define an *instance* of the transitive-closure problem to be a 4-tuple $I = (s, d, n, e)$, representing the arguments of the problem $S(s, d, n, e)$. Then,

define a *strategy* for solving instance I with time limit t to be a tree with the following properties.

(1) Every node is labeled by an instance.

(2) The root is labeled I.

(3) Each interior node is also labeled by a recursive rule, R1, R2, R3, or R4. Optionally, we attach the value of the key parameter, s_1 or k, to help the reader follow what is going on.

(4) Each leaf is additionally labeled by a basis rule, B1 or B2.

(5) The interior nodes make sense, in that if interior node v is labeled by instance J and rule Ri, then the children of v are labeled by the instance(s) found on the right of the arrow in the definition of Ri, as given in § 4.

(6) The leaves make sense, in that if a leaf is labeled by instance (s, d, n, e) and rule B2, then $d \leqq t$. There is no similar constraint for leaves labeled B1, since that algorithm can be applied in polylog time.

Example 6.1. Suppose $I = (1, n, n, e)$, that is, single-source, sparse transitive-closure. Let $t = n^{1/3}$. The solution provided by Algorithm 5.2 is the tree of Fig. 6.2, since Case 5, as enumerated in Fig. 5.3, applies. Note that we have simplified expressions in some of the instances by dropping terms that are dominated by other terms. For example, at the rightmost child of the root, the fourth argument, $e + n^{2/3}$, has been simplified to e.

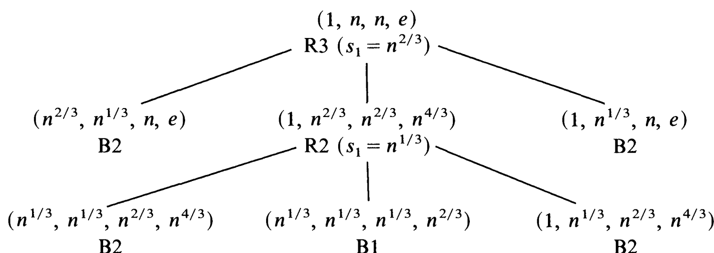


FIG. 6.2. *Solution for single-source transitive-closure with* $t = n^{1/3}$.

The *cost* of a solution is defined recursively, up the tree. As in previous sections, costs elide factors of log n. For a leaf, we compute the work for the associated basis rule, as given in § 4: n^3 for B1 and $es + ed$ for B2. The cost for an interior node is the sum of the costs of its children, if R1, R2, or R3 is used, and k times the cost of its lone child if R4 is used with parameter k.[7]

The following function $f(I, t)$ summarizes the cost of the strategy generated by Algorithm 5.2, as we shall prove. Define

$$f(I, t) = es + e \min(d, \sqrt{n}) + \alpha_1 \left[e\frac{n}{t} + s\frac{n^2}{t^2} \right] + \alpha_2 \left[\frac{n^3}{t^4} \right]$$

where $\alpha_1 = 1$ if $d > t$ and $\alpha_1 = 0$ otherwise; $\alpha_2 = 1$ if $d > t^2$, and $\alpha_2 = 0$ otherwise. We shall show that $f(I, t)$ is a lower bound on the cost of any strategy for $I = (s, d, n, e)$ with time limit t. First, we show that $f(I, t)$ accurately reflects the cost of the strategy generated by Algorithm 5.2.

[7] We shall see that R4 can never be used to advantage, but since it is a plausible strategy, we need to consider its use.

LEMMA 6.3. *The work of Algorithm* 5.2*'s strategy, as given in Fig.* 5.3, *is equal to that of the function* $f(I, t)$ *above.*

Proof. We divide the proof according to the relationship between d and t.

Case A. $d \leqq t$. Then $\alpha_1 = \alpha_2 = 0$, and $f(I, t)$ is $es + e \min(d, \sqrt{n})$. Also, only Cases 1 and 2 of Fig. 5.3 can apply. In Case 1, where $d \leqq \sqrt{n}$, $\min(d, \sqrt{n}) = d$, so $f(I, t) = es + ed$; this formula is exactly that given by Fig. 5.3. In Case 2, $d > \sqrt{n}$, so $f(I, t) = es + e\sqrt{n}$, also as in Fig. 5.3.

Case B. $t < d \leqq t^2$. Then $\alpha_1 = 1$, but $\alpha_2 = 0$. Only Cases 2–4 of Fig. 5.3 can apply. In Case 2, $d > \sqrt{n}$, so $f(I, t)$ is

$$es + e\sqrt{n} + en/t + sn^2/t^2.$$

Since $t \geqq \sqrt{n}$ in Case 2, $en/t \leqq e\sqrt{n}$, so the second term dominates the third. Also, $sn^2/t^2 \leqq sn$, so the first term dominates the fourth. Thus, the above formula for $f(I, t)$ reduces to $es + e\sqrt{n}$, as given for Case 2 in Fig. 5.3.

In Cases 3 and 4, we do not know how d and $\sqrt{n}$ compare, so we write $f(I, t)$ as

$$es + e \min(d, \sqrt{n}) + en/t + sn^2/t^2.$$

In Case 3, we have $s \geqq n/t$, so the first term dominates the third. Also, $t < \sqrt{n}$, so $s \geqq n/t > \sqrt{n}$. Hence, $es > e\sqrt{n}$, and the first term dominates the second. Thus, $f(I, t)$ simplifies to the formula for Case 3 in Fig. 5.3.

Finally, consider Case 4. There, $s < n/t$, so the third term dominates the first. Also, $t < \sqrt{n}$, so $en/t > e\sqrt{n}$, and the third term dominates the second. Thus, $f(I, t)$ reduces to the formula given by Case 4 of Fig. 5.3.

Case C. $d > t^2$. Then $\alpha_1 = \alpha_2 = 1$, and $f(I, t)$ is

$$es + e \min(d, \sqrt{n}) + en/t + sn^2/t^2 + n^3/t^4.$$

Here, only Cases 3 or 5 of Fig. 5.3 can apply. If Case 3 of Fig. 5.3 applies, we can argue as in Case B above that the first term dominates the second and third. Also, since $s \geqq n/t$, the fourth term is at least n^3/t^3, and so dominates the fifth term. Thus, $f(I, t)$ simplifies to the formula of Case 3 in Fig. 5.3, that is, the first and fourth terms, above.

If Case 5 holds, then $s < n/t$ and $t < \sqrt{n}$ tell us that the third term dominates the first and second, which reduces the above formula to that of Case 5 in Fig. 5.3. □

LEMMA 6.4. *Without loss of generality, we may assume that the optimal strategy does not use* R4.

Proof. Note that $f(I, t)$ never grows more than linearly with s. Each term is either independent of s or linear in s. In particular, α_1 and α_2 are independent of s, and so cannot, by their discontinuity, make $\partial f/\partial s$ exceed one. Thus, if a strategy uses R4 with parameter k at an interior node v, and that node has child with label $I = (s, d, n, e)$, we can eliminate that child and multiply the number of sources by k in each descendant whose number of sources depends on s. Note that the validity of a strategy can only depend on the relationship between the distance and the time limit, so the strategy cannot be rendered invalid by this transformation. We have multiplied the cost of each descendant of v by at most k, and thus the cost of v itself has not increased, since we no longer have to multiply the cost of v's child by k to get the cost of v. □

THEOREM 6.5. *The cost of any strategy for solving* $I(s, d, n, e)$ *in time* t *is* $\tilde{\Omega}(f(I, t))$.

Proof. By Lemma 6.4 we may assume that the strategy does not use R4. We proceed by induction on the height of the tree T for the strategy in question.

Basis. If the tree T is a leaf labeled B1, then the cost of the strategy is n^3, and we note that no term of $f(I, t)$ exceeds n^3. If the leaf is labeled B2, then we must have

$d \leqq t$. Thus, $\alpha_1 = \alpha_2 = 0$, and $f(I, t) = es + e \min(d, \sqrt{n})$. The cost of T is $es + ed$, which cannot be less than $f(I, t)$.

Induction. The root of T is labeled by R1, R2, or R3. We shall consider each of the five possible terms of $f(I, t)$: es, $e \min(d, \sqrt{n})$, en/t, sn^2/t^2, and n^3/t^4, and show that, when that term is present in the formula for f (i.e., the appropriate α is one, if necessary), then one of the subproblems for the rule used also has at least that term.

(1) Term es. Each of the rules R1, R2, and R3 contains on the right an instance with s sources and at least e arcs. By the inductive hypothesis, the cost of that subproblem is bounded below by f, and so has a term $e's$ for some $e' \geqq e$.

(2) Term $e \min(d, \sqrt{n})$. On the right-hand side of each recursive rule, the first subproblem has s_1 sources, distance n/s_1, n nodes, and e arcs. By the induction hypothesis, the cost of this subproblem is at least $es_1 + e \min(n/s_1, \sqrt{n})$, which is at least $e\sqrt{n}$, because either $s_1 \geqq \sqrt{n}$ or $n/s_1 \geqq \sqrt{n}$.

(3) Term en/t. Here, we may assume $\alpha_1 = 1$, that is, $d > t$. We consider the first subproblem of each rule, which is $(s_1, n/s_1, n, e)$, and branch depending on whether $n/s_1 \leqq t$.

(a) $n/s_1 \leqq t$. Then the cost of the first subproblem includes, by the inductive hypothesis, a term es_1. Since $n/s_1 \leqq t$, this term is at least en/t.

(b) $n/s_1 > t$. Since n/s_1 is the distance in the first subproblem, the value of f for that subproblem has $\alpha_1 = 1$, and thus includes term en/t.

(4) Term sn^2/t^2. We distinguish the same two subcases as in (3). In Case (a), where $n/s_1 \leqq t$, note that each rule has a subproblem with s sources and at least s_1^2 arcs; this subproblem is the last in R1 and R2, and the second in R3. The cost for this subinstance includes a term $s_1^2 s$, which is at least sn^2/t^2, if Case (a) holds. In Case (b), the last subproblem of each rule has s sources, n nodes, and distance at least n/s_1, which exceeds t in Case (b). Thus, $\alpha_1 = 1$ for this subproblem, and its cost includes a term sn^2/t^2 by the inductive hypothesis.

(5) Term n^3/t^4. Here, we may assume $\alpha_1 = \alpha_2 = 1$. There are three subcases, depending on how n/s_1 compares with t and t^2.

(a) If $n/s_1 > t^2$, then for the first subproblem of each rule, the value of α_2 is one. Thus, the cost of this subproblem includes term n^3/t^4 by the inductive hypothesis.

(b) If $t < n/s_1 \leqq t^2$, then α_1 for the first subproblem is one, and the cost of that subproblem includes a term $s_1 n^2/t^2$. Since $n/s_1 \leqq t^2$ is assumed, we have $s_1 \geqq n/t^2$, and therefore $s_1 n^2/t^2 \geqq n^3/t^4$.

(c) If $n/s_1 \leqq t$, then $ds_1/n \geqq d/t > t$; the latter inequality follows because $\alpha_2 = 1$ for the instance at hand, or else we would not have to deal with the term n^3/t^4 at all. For each of rules R1, R2, and R3, the second subproblem, call it $S(s', d', n', e')$, satisfies $n' \geqq s_1$, $e' \geqq s_1^2$, and $d' \geqq ds_1/n > t$. By the latter, the α_1 bit for this subproblem is one, and its cost includes the term $e'n'/t \geqq s_1^3/t \geqq n^3/t^4$. □

COROLLARY 6.6. *The strategy of Algorithm* 5.2 *is optimal.*

Proof. The cost of that strategy is exactly that given by $f(I, t)$, as we learned from Theorem 5.4 and Lemma 6.3. □

7. Important special cases. The most important instances of the general transitive closure are:

(1) $S(1, n, n, e)$, or sparse, single-source.

(2) $S(n, n, n, e)$, or sparse, all-pairs.

(3) $S(1, n, n, n^2) = D(1, n, n)$, or dense, single-source.
(4) $S(n, n, n, n^2) = D(n, n, n)$, or dense, all pairs.
All these have d, the distance, equal to n, the number of nodes, and have s, the number of sources, equal to one or n.

All-pairs problems. When we let $s = d = n$ in the formula $f(I, t)$ of the previous section, we are left with the expression

$$en + e\sqrt{n} + \alpha_1(en/t + n^3/t^2) + \alpha_2 n^3/t^4.$$

Also, we note that since $d = n$, $\alpha_1 = 1$ as long as $t < n$, which is the interesting case, and $\alpha_2 = 1$ whenever $t < \sqrt{n}$. We may therefore drop the last term, $\alpha_2 n^3/t^4$, since whenever it is nonzero, it is dominated by the term $\alpha_1 n^3/t^2$. Moreover, the second and third terms are clearly dominated by the first, so we can write

$$f(n, n, n, e, t) = en + n^3/t^2. \tag{7.1}$$

Note that we do not need the factor α_1 on the second term, since whenever $\alpha_1 = 0$, n^3/t^2 is no larger than n, and therefore is dominated by the first term anyway. Figure 7.2(a) gives the strategy for the all-pairs case, when $t \geqq \sqrt{n}$; here, Case 2 of Fig. 5.3 applies, and the cost is en. Figure 7.2(b) shows the strategy when $t < \sqrt{n}$; here Case 3 of Fig. 5.3 pertains.

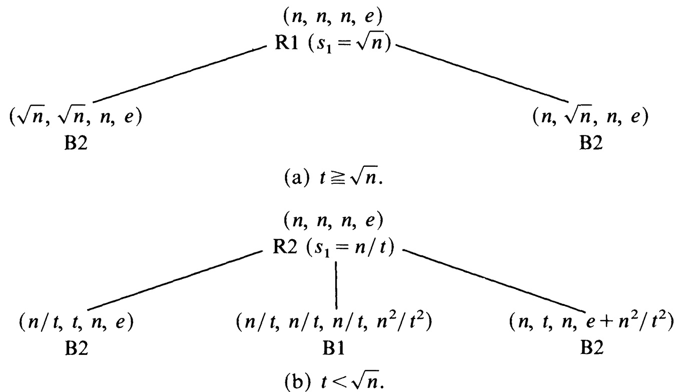


FIG. 7.2. *Strategies for the all-pairs problem.*

A useful simplification is to assume that $t = n^{\varepsilon}$ for some small ε, and $e = n^{\alpha}$, for some α between one and two. In (7.1), the en term dominates as long as $\alpha \geqq 2 - 2\varepsilon$. We can thus solve the all-pairs problem optimally, as long as the graph is not too sparse. For time $\sqrt{n}$, that is, $\varepsilon = \frac{1}{2}$, there is no constraint on e, but as the time shrinks, the number of arcs must increase if the algorithm is to be optimal, until as the time approaches polylog time, the graph must be dense.

The dense, all-pairs problem. In the dense case, where $e = n^2$, (7.1) reduces to n^3. That is, our algorithms offer no help for the dense, all-pairs problem. In that case, the basis rule B1 is as good a strategy as we know.

Single-source problems. Now, let $s = 1$ and $d = n$ in the formula $f(I, t)$. The resulting formula is

$$e + e\sqrt{n} + \alpha_1(en/t + n^2/t^2) + \alpha_2 n^3/t^4.$$

Again, $\alpha_1 = 1$ as long as $t < n$, and $\alpha_2 = 1$ whenever $t < \sqrt{n}$. Now, note that n^2/t^2 can never exceed en/t, so we may eliminate the fourth term. The first term is clearly dominated by the second. We can thus write

$$f(1, n, n, e, t) = e\sqrt{n} + en/t + n^3/t^4. \qquad (7.3)$$

We do not need multipliers α_1 or α_2 on the last two terms, because when one of the α's is zero, its term is always dominated by the first term.

Figure 7.4(a) shows the strategy of Algorithm 5.2 for the single-source problem when $t \geqq \sqrt{n}$. It represents an alternative to Algorithm 2.3, which was defined only for the case $t = \sqrt{n}$, but evidently works for larger t.

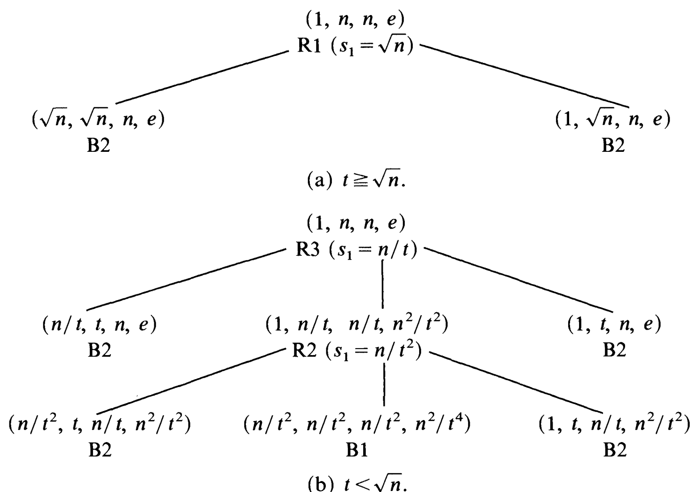


FIG. 7.4. *Strategies for the single-source problem.*

If we assume $t = n^\varepsilon$ for $\varepsilon \leqq \frac{1}{2}$, then the $e\sqrt{n}$ term in (7.3) can be dropped. The term en/t dominates as long as $e = n^\alpha$, and $\alpha \geqq 2 - 3\varepsilon$. Put another way, we can solve the problem with time $t = \tilde{O}(n^\varepsilon)$ and work $\tilde{O}(en^{1-\varepsilon})$ (i.e., $en^{1-2\varepsilon}$ processors), as long as $\varepsilon \geqq (2-\alpha)/3$. For example, even in the sparsest case, $\alpha = 1$, we can use $\tilde{O}(n^{1/3})$ time and $\tilde{O}(n^{4/3})$ processors.

The dense, single-source problem. For the dense case, (7.3) reduces to $n^{2.5} + n^3/t$, or just n^3/t if we assume that only $t \leqq \sqrt{n}$ is of interest. Thus, for the single-source problem, even in the dense case we offer some improvement over the obvious solution of applying B1; the more time we are willing to take, up to $t = \sqrt{n}$, the more improvement in the total work we achieve. Note, however, that if t is polylog in n, then we achieve nothing, since n^3/t stands for $\tilde{O}(n^3/t)$, which is $\tilde{O}(n^3)$ in this case. Thus, B1 is still the best $\mathcal{NC}$ algorithm we know for the dense, single-source problem.

8. Extension to breadth-first search. Ideally, we would like the techniques described here to work for generalized transitive closure, such as shortest paths, or even general closed semirings, as in Aho, Hopcroft, and Ullman [1974]. We cannot do so, principally because the efficiency associated with basis rule B2 depends on each node being reached with only one fact about any distinguished node. However, we can push our techniques somewhat further. Recall that the single-source BFS (breadth-first-search) problem of Gazit and Miller [1988] is to find, for each node v, the least

number of arcs on the paths from the source node to v. Alternatively, we can view BFS as the special case of the shortest-path problem when the arcs all have unit length. We can solve the single-source BFS problem in the time given by (7.3); it is not known whether the same holds for the all-sources BFS problem and formula (7.1).

We shall discuss only the hard (and interesting) case where $t \leqq \sqrt{n}$. The strategy we use is similar to that illustrated in Fig. 7.4(b) for single-source transitive closure. First, we select $\tilde{O}(n/t)$ "distinguished nodes" and explore distance t from them. Then, we select from among the distinguished nodes $\tilde{O}(n/t^2)$ "superdistinguished nodes," and solve a restricted search problem from these. There are sufficiently few superdistinguished nodes that we can compute all shortest paths among them with work $(n/t^2)^3 = n^3/t^6$, by conventional means.[8]

As we remarked in § 2, the technique for step (2) of Algorithm 2.3 is not essential, except to save a factor of log n. We can instead use e/t processors to search forward from any one distinguished node for distance t in time $\tilde{O}(t)$, as follows. The algorithm is essentially a breadth-first-search, in which we divide the graph into layers according to the length of the shortest path from the given source. Suppose we have constructed the first i layers and marked all nodes that have been reached.

(1) The arcs out of the nodes on layer i are divided evenly among the e/t processors.

(2) Each processor finds the unmarked nodes reached by these arcs, and marks them as belonging to layer $i+1$. The reached nodes become layer $i+1$.

(3) A list of the nodes on layer $i+1$ is made, eliminating duplicates, and a count of the arcs out of this layer is made, to facilitate step (1) for layer $i+1$.

Starting with only the given distinguished node in layer zero, we can construct layers up to t.

LEMMA 8.1. *The above construction correctly computes the length of the shortest path from the given source w to each node that is at distance at most t from w. It takes parallel time $\tilde{O}(t)$.*

Proof. The correctness should be evident. Subtlety is in the running time. Intuitively, each arc is explored only once, so while we may spend a lot of time processing one layer, the total time spent on all the layers is $\tilde{O}(t)$. Let e_i be the number of arcs out of the nodes at layer i, and note that $\sum_{i=0}^{t} e_i \leqq e$. The time spent processing layer i is easily seen to be $\tilde{O}(e_i t/e)$; the principal costs are exploring the arcs in step (2) and sorting in step (3). But

$$\sum_{i=0}^{t} e_i t/e \leqq (t/e) \sum_{i=0}^{t} e_i \leqq t.$$

Thus, the total time over all layers is $\tilde{O}(t)$. □

Note that the total work performed by the algorithm outlined above is $\tilde{O}(en/t)$. That is, there are $\tilde{O}(n/t)$ distinguished nodes, with e/t processors assigned to each. The total number of processors, en/t^2, is multiplied by the time, $\tilde{O}(t)$, to get the total work.

Search from superdistinguished nodes. If we pick $\tilde{O}(n/t)$ distinguished nodes at random, and include the single source among them, we can use the above algorithm to search from each distinguished node with work $\tilde{O}(en/t)$ and time $\tilde{O}(t)$. The result

[8] Note that we cannot do so for the graph of the distinguished nodes, because that would take $(n/t)^3$ work, which is larger than the n^3/t^4 term of (7.3). However, one more level, to the superdistinguished nodes, gets us down to work n^3/t^6, which can be neglected when compared with the n^3/t^4 term.

will tell us all the paths from one distinguished node to another, provided that path is of length t or less. More formally, if we have discovered that searching from distinguished node w_1, we can reach distinguished node w_2, and the layer of w_2 with respect to w_1 is i, then $i \leqq t$, and i is truly the length of a shortest path from w_1 to w_2. Conversely, if we have not discovered a path from w_1 to w_2, then the length of a shortest path from w_1 to w_2, if it exists, is greater than t.

We could use a conventional path-doubling-by-matrix-multiplication algorithm to find the lengths of the shortest paths between each pair of distinguished nodes, in the graph of distinguished nodes whose arcs are weighted by the lengths of the paths found between nearby distinguished nodes. However, that would take work $\tilde{O}(n^3/t^3)$, and we need to do better, specifically $\tilde{O}(n^3/t^4)$.

Thus, we construct the graph H of distinguished nodes with weighted arcs representing shortest paths up to length t, but before proceeding, we select at random $\tilde{O}(n/t^2)$ superdistinguished nodes for this graph. We also include the original source node among the superdistinguished nodes. We shall compute the lengths of shortest paths between every pair of superdistinguished nodes first. Note that arcs in H have weights up to t. "Shortest" refers to weighted path length, not to the number of arcs. However, paths in H represent paths in the original graph, and the weighted path length in H equals the length (number of arcs) on the corresponding paths of the original graph.

The idea is illustrated in Fig. 8.2. We start from some superdistinguished node x, and look at a shortest path in H from x to some (distinguished) node y. We divide the nodes along the path, all of which are distinguished, into layers according to the length of their shortest path from x; the layers each have width t; that is, the first layer consists of nodes with shortest paths from x of length up to t, the next of nodes with shortest paths from $t+1$ to $2t$, and so on. There are n/t layers, since each path among the distinguished nodes represents a path in the original graph of n nodes, and therefore has length at most n. However, we shall only want to consider the first t layers, and thus discover paths of length up to t^2 in the original graph; as we shall see, these paths are represented by paths with at most $2t$ arcs in the graph H.

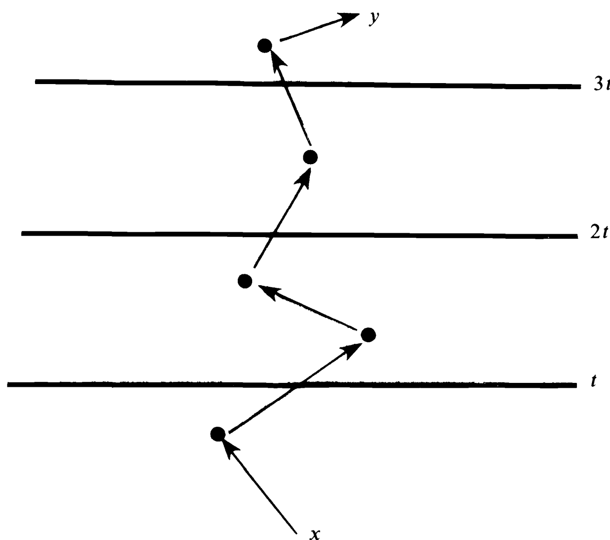


FIG. 8.2. *Shortest weighted path from x to y.*

When we performed a breadth-first-search in the original graph of n nodes, we could proceed layer by layer, since the arcs were unweighted, and each layer could be constructed directly from the one below. We are not so fortunate in the graph of distinguished nodes, since now arcs have weights up to t, and even if we regard layers as having width t, as suggested by Fig. 8.2, we cannot compute each layer directly from the one below. Fortunately, we can do so in two passes, as the next lemma argues.

LEMMA 8.3. *In the graph of distinguished nodes, a shortest path from x to y may be assumed not to have three consecutive nodes in one layer.*

Proof. Suppose w_1, w_2, and w_3 are three consecutive (distinguished) nodes belonging to the same layer, and lying on a shortest path from x to y. Then the path from w_1 to w_3 is not longer than t, and by Lemma 8.1, the search for distance t from distinguished node w_1 in the original graph reached w_3. Thus, H has an arc $w_1 \to w_3$, whose weight cannot be greater than the sum of the weights of the arcs $w_1 \to w_2$ and $w_2 \to w_3$, since all such arcs are weighted with true shortest path lengths. It follows that we could have eliminated w_2 from the shortest path from x to y. □

Thus, we can build the shortest-path information layer by layer, if we use two passes per layer, one to discover nodes of that layer whose predecessor on the shortest path from a given superdistinguished node is in the previous layer, and a second to discover nodes with one previous node on the shortest path that is in the same layer. Of course, no node can have a shortest path where the previous node is not in the same or next-lower layer, because arc weights are limited by t.

(1) For each node w in layer i, whose distance from superdistinguished node x has been calculated to be j (thus, $(i-1)t<j\leqq it$), consider each arc in the graph of distinguished nodes out of w. If j plus the weight d of that arc exceeds it, the arc enters node u, and u is not in a layer below $i+1$, then generate a fact that there is a path from x to u of length $j+d$. Note that $j+d\leqq(i+1)t$, so u belongs to layer $i+1$.

(2) Sort the generated facts by target node, and eliminate distances for each node other than the minimum. Note that the resulting distance for a node u in layer $i+1$ may not yet be minimum, since there may be another node in layer $i+1$ that the shortest path from x to u goes through. However, if u is a node of layer $i+1$ such that the shortest path from x to some other node v in layer $i+1$ goes through u, then the shortest path to u is already correct, or else the shortest path from x to v goes through three nodes in layer $i+1$, contradicting Lemma 8.3.

(3) Examine each node u already in layer $i+1$; say the current estimate of the length of the shortest path from x to u is k. For each arc out of u, of weight c and reaching v, generate the fact that there is a path from x to v of weight $k+c$, provided $k+c\leqq(i+1)t$.

(4) Again sort all the facts, including those remaining from step (2), according to their target node, and for each target node select the shortest path for that node. The result is that all nodes of layer $i+1$ are found and have the length of their shortest path from x calculated.

The complete algorithm for searching from each of the $\tilde{O}(n/t^2)$ superdistinguished nodes for weighted distance t^2 is to use n^2/t^3 processors for each of the $\tilde{O}(n/t^2)$ superdistinguished nodes, and use them to perform the above search for t layers. Thus, all paths between superdistinguished nodes of length at most t^2 are found. Note that the length t^2 can be thought of either as weighted path length in H, or as number of arcs in the original graph.

LEMMA 8.4. *The above steps correctly compute all shortest paths among superdistinguished nodes, provided those paths are of length at most t^2. Furthermore, the time taken is $\tilde{O}(t)$, and the total work is $\tilde{O}(n^3/t^4)$.*

Proof. Correctness follows from Lemma 8.3. For now, note that there are $\tilde{O}(n/t)$ nodes in the graph of distinguished nodes, and therefore at most $\tilde{O}(n^2/t^2)$ arcs. Each arc is considered at most twice, in steps (1) and (3). Thus, by an argument similar to that of Lemma 8.1, we can conclude that the time taken by the n^2/t^3 processors assigned to each superdistinguished node as a source, summed over each of the t layers, is at most $\tilde{O}(t)$. Since there are $\tilde{O}(n/t^2)$ superdistinguished nodes, the total number of processors is n^3/t^5, and when we multiply this figure by the time taken, we get total work of $\tilde{O}(n^3/t^4)$. □

The complete algorithm. We can now put together the ideas summarized in Lemmas 8.1 and 8.4 to get an algorithm that uses work $en/t+n^3/t^4$.[9]

ALGORITHM 8.5. Single-source breadth-first search.
INPUT: A graph G of n nodes and e arcs, and a source node v_0. Also, a time limit $t \leqq \sqrt{n}$.
OUTPUT: For each node v in G, the length of (number of arcs on) a shortest path from v_0 to v.
METHOD: We perform each of the following steps.
(1) Pick $\tilde{O}(n/t)$ distinguished nodes at random, and add v_0 to the set of distinguished nodes, whether or not it was picked.
(2) Perform a breadth-first search from each distinguished node for distance t, by the technique outlined prior to Lemma 8.1.
(3) Construct graph H whose nodes are the distinguished nodes selected in step (1). There is an arc $u \to v$ in H, weighted d, if in step (2) it was determined that a shortest path from u to v is of length $d \leqq t$.
(4) Select $\tilde{O}(n/t^2)$ superdistinguished nodes in H, and include v_0 among them, whether or not it was picked.
(5) Explore H from each superdistinguished node for t layers (weighted distance t^2), using the technique outlined prior to Lemma 8.4.
(6) Construct a graph J of the superdistinguished nodes, with weighted arcs as discovered in (5). Compute shortest paths in J by path-doubling-by-matrix-multiplication, using $\tilde{O}(t)$ time and $\tilde{O}(n^3/t^6)$ work, since the number of nodes is $\tilde{O}(n/t^2)$. Since v_0 is among the nodes of J, we now have the length of the shortest path from v_0 to each superdistinguished node, which we may regard as a vector of length $\tilde{O}(n/t^2)$.
(7) Treat the information obtained in step (5), giving shortest paths of length up to t^2 from the superdistinguished nodes to all distinguished nodes as an $\tilde{O}(n/t^2)\times\tilde{O}(n/t)$ matrix. Multiply this matrix by the vector from step (6), using + for scalar multiplication and min for scalar addition, thus obtaining the length of the shortest path from v_0 to every distinguished node. The obvious algorithm can be performed in time $\tilde{O}(t)$ and work $\tilde{O}(n^2/t^3)$. Again, regard this information as a vector, this time of length $\tilde{O}(n/t)$.
(8) Treat the information obtained in step (2), giving shortest paths of length up to t from the distinguished nodes to all nodes, as an $\tilde{O}(n/t)\times n$ matrix. Multiply this matrix by the vector from step (7), again using + for scalar multiplication and min for scalar addition. The result is the length of a shortest path from v_0 to every node. The obvious algorithm can be performed in $\tilde{O}(t)$ time and $\tilde{O}(n^2/t)$ work.

THEOREM 8.6. *Algorithm 8.5 is correct with high probability.*

Proof. If we pick $(cn \log n)/t$ distinguished nodes, for sufficiently high c, we can make the probability as high as we like that there is a distinguished node within

[9] As mentioned, we assume $t \leqq \sqrt{n}$. Thus, the term $e\sqrt{n}$ in (7.3) can be neglected.

every t nodes along a shortest path from v_0 to any node. Likewise, by picking $(cn \log n)/t^2$ superdistinguished nodes, we can assume with high probability that there is a superdistinguished node at least every t^2 nodes along every shortest path. Consequently, with high probability, the transitive-closure J computed at step (6) is correct, and at step (7), we correctly have the length of a shortest path from v_0 to every superdistinguished node. We may regard any shortest path as traveling from v_0 to some superdistinguished node (perhaps v_0 itself), then through a distance less than t^1, through at most $2t$ distinguished nodes, and finally through nondistinguished nodes for a distance of at most t. Step (7) takes us from the last superdistinguished node to the last distinguished node, and step (8) takes us the rest of the way. □

THEOREM 8.7. *Algorithm* 8.5 *takes time* $\tilde{O}(t)$ *and work* $\tilde{O}(en/t+n^3/t^4)$.

Proof. It is easy to check that each step can be done within $\tilde{O}(t)$ time. Steps (1) and (4) can be done with $\tilde{O}(n)$ work, as discussed in connection with Theorem 3.4. The work of step (2) is $\tilde{O}(en/t)$ by Lemma 8.1. Step (3) is easily seen to take $\tilde{O}(n^2/t^2)$ work. Step (5) requires $\tilde{O}(n^3/t^4)$ work by Lemma 8.4. Step (6) takes $\tilde{O}(n^3/t^6)$ work, step (7) uses $\tilde{O}(n^2/t^3)$, and step (8) requires $\tilde{O}(n^2/t)$, as discussed in Algorithm 8.5. It is easy to check that each of these terms is dominated by either en/t or n^3/t^4, assuming $t \leqq \sqrt{n}$. □

9. Open problems. We may have made some progress toward the real goal of deterministic, optimal parallel algorithms for single-source and sparse cases of transitive-closure and related problems, such as shortest paths. There is a long sequence of open problems suggested by these results. We enumerate some of them here.

(1) Can the algorithms be made not to depend on probabilistic choice: that is, are there deterministic algorithms with the same performance?

(2) Can we reduce the work still further, using a one-sided-error, high-probability algorithm, like the ones presented here?

(3) Can we generalize the algorithm for sparse, all-pairs transitive-closure to the breadth-first-search problem? Note that work $en+n^3/t$ is achievable, which is better than the obvious algorithm, but does not match the bound of (7.1).

(4) Does any of this material generalize to the general shortest-path problem, where arcs initially have arbitrary weights?

Acknowledgments. We were helped by discussions concerning this material with Amotz Bar-Noy, Joan Feigenbaum, Seffi Naor, and Vijay Vazirani. The use of computer equipment belonging to Princeton University is acknowledged by the first author.

REFERENCES

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

P. A. BLONIARZ, M. J. FISCHER, AND A. R. MEYER [1976], *A note on the average time to compute transitive closure*, 3rd Internat. Colloquium on Automata, Languages, and Programming, Edinburgh, U.K., Springer-Verlag, Berlin, New York.

R. P. BRENT [1974], *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21, pp. 201–208.

A. Z. BRODER, A. R. KARLIN, P. RAGHAVAN, AND E. UPFAL [1989], *Trading space for time in undirected s-t connectivity*, in Proc. 21st annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York.

R. COLE AND U. VISHKIN [1986], *Approximate and exact parallel scheduling with applications to list, tree, and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 478–491.

D. COPPERSMITH AND S. WINOGRAD [1987], *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, pp. 1-6.

H. GAZIT [1986], *An optimal randomized parallel algorithm for finding connected components in a graph*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 492-501.

H. GAZIT AND G. L. MILLER [1988], *An improved parallel algorithm that computes the* BFS *numbering of a directed graph*, Inform. Process. Lett., 28, pp. 61-65.

D. H. GREENE AND D. E. KNUTH [1982], *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston.

C. P. SCHNORR [1978], *An algorithm for transitive closure with linear expected time*, SIAM J. Comput., 7, pp. 127-133.

Y. SHILOACH AND U. VISHKIN [1982], *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3, pp. 57-63.

K. SIMON [1986], *An improved algorithm for transitive closure on acyclic digraphs*, 13th Internat. Colloquium on Automata, Languages, and Programming, Rennes, France, Springer-Verlag, Berlin, New York, pp. 376-387.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 126-143, February 1991
© 1991 Society for Industrial and Applied Mathematics
007

COMPLEXITY OF SENTENCES OVER NUMBER RINGS*

SHIH PING TUNG†

Abstract. Let I be an algebraic integer ring. The decision problem of the solvability of diophantine equations with parameters in I is proved to be co-NP-complete. However, the decision problem of the solvability of diophantine equations with parameters in all algebraic integer rings is in P. Let $\varphi(x, y)$ be a quantifier-free arithmetical formula in conjunctive normal form or disjunctive normal form. The decision problem of the sentences of the form $\exists x \forall y\, \varphi(x, y)$, true in I, is NP-complete. Then the decision problem of the sentences of the form $\forall x \exists y\, \varphi(x, y)$, true in I, is co-NP-complete, whereas the decision problem of the sentences of the form $\forall x \exists y\, \varphi(x, y)$, true in all algebraic integer rings, is in P. Some other related decision problems are also proven to be in P.

Key words. arithmetical sentences, diophantine equations with parameters, factorization of polynomials, algebraic integer rings, P, NP-completeness

AMS (MOS) subject classifications. 03B25, 68Q25, 11R04

Introduction. In this paper, we will study the computational complexity of some decision problems on arithmetical sentences over algebraic integer rings. These problems arise quite naturally, but previously we did not even know that these problems were decidable.

The language L, used in this paper, contains all the usual logical symbols: $\wedge$ (and), $\vee$ (or), $\neg$ (negation), $\Rightarrow$ (implies), $\Leftrightarrow$ (if and only if), $\exists$ (there exists), $\forall$ (for every), $=$ (equals), with constants over $\mathbf{Z}$, variables, and arithmetical symbols $+$ (addition) and $\cdot$ (multiplication). An arithmetical sentence is a sentence constructed from the language L. The arithmetical theory of a ring R is the set of all arithmetical sentences true in the ring R. The arithmetical theory of a class of rings is the set of all arithmetical sentences that are true in every ring of the class. An arithmetical theory is decidable if there is an algorithm to determine whether a given arithmetical sentence is in this set or not. If there is no such algorithm, then the theory is undecidable.

By Godel's incompleteness theorem, the arithmetical theories of the set of non-negative integers $\mathbf{N}$ and the ring of rational integers $\mathbf{Z}$ are undecidable. Robinson [21] proved that the arithmetical theory of the rational number field $\mathbf{Q}$ is undecidable. Later she proved that the arithmetical theories of an algebraic number field and an algebraic integer ring are undecidable [22]. Robinson [23] also proved that the arithmetical theories of some other rings of algebraic integers are undecidable.

Hilbert's tenth problem asks if there is a decision procedure so that given a diophantine equation, we can tell whether this equation is solvable in $\mathbf{Z}$ or not. This is equivalent to asking the decidability of the true sentences in $\mathbf{Z}$ with the form $\exists x_1 \cdots \exists x_n f(x_1, \cdots, x_n) = 0$ for the arbitrary polynomial $f(x_1, \cdots, x_n)$ over $\mathbf{Z}$. From the work of Davis, Putnam, Robinson, and Matijasevic, this problem was shown to be undecidable (cf. [3]). Denef and Lipshitz [4], [5], [6] extended this undecidable result to some rings of algebraic integers. Having all these undecidable results, we can then ask what subsets of these theories are decidable. And we can further ask what the computational complexities of these decidable cases are.

* Received by the editors July 23, 1986; accepted for publication (in revised form) March 8, 1990. This research was partially supported by the National Science Council of the Republic of China under grant NSC-76-0208-M-033-06.

† Department of Mathematics, Chung Yuan Christian University, Chung Li, Taiwan 32023, Republic of China.

It was shown [18] that for binary quadratics of the form $ax^2+by=c$, a, b, $c\in \mathbf{N}$, the problem of deciding whether it is solvable in $\mathbf{N}$ is NP-complete. Manders and Adleman also showed that the problem of deciding whether, for a, b, c of $\mathbf{N}$, $x^2\equiv a \pmod b$, $0\leqq x\leqq c$, has a solution in $\mathbf{N}$ is NP-complete. Those decision problems can be easily shown in NP, because there is a bound on the size of possible solutions to either problem given by a polynomial in the coefficients a, b, and c. Baker has shown that various diophantine equations in two unknowns are decidable [1]. But the decision problem of diophantine equations in two unknowns, in general, is still open.

Given a polynomial $f(x_1,\cdots,x_n,y)$, we may ask whether or not for any n numbers $x_1',\cdots,x_n'$ the equation $f(x_1',\cdots,x_n',y)=0$ is solvable, or equivalently, the sentence $\forall x_1\cdots\forall x_n\exists y f(x_1,\cdots,x_n,y)=0$ is true. This problem is called diophantine equations with parameters in number theory [24]. The decision problem of the solvability of diophantine equations with parameters over $\mathbf{N}$ is undecidable [27]. This follows quite easily from the negative answer to Hilbert's tenth problem over $\mathbf{N}$. We do not know the answer to this decision problem over $\mathbf{Q}$, but a positive answer to this decision problem implies a positive answer to Hilbert's tenth problem over $\mathbf{Q}$. This is still an outstanding open problem. However, in [27] we proved that the solvability of diophantine equations with parameters over an algebraic integer ring is decidable. In this paper, we will prove that this decision problem is co-NP-complete if the number of parameters is bounded by a number greater than 0.

We call φ an $\forall\exists$ or $\exists\forall$ sentence if and only if φ is an arithmetical sentence logically equivalent to a sentence of the form $\forall x\exists y\,\psi(x,y)$ or $\exists x\forall y\,\psi(x,y)$, respectively, where $\psi(x,y)$ is a formula containing no quantifiers and no other free variables except x and y. A formula ψ is in conjunctive normal form (CNF) if it has the form $\psi=\psi_1\wedge\psi_2\wedge\cdots\wedge\psi_n$, where $\psi_1=\varphi_{i,1}\vee\varphi_{i,2}\vee\cdots\vee\varphi_{i,m_i}$ and each $\varphi_{i,j}$ is an equation $f=0$ or inequation $g\neq 0$. A disjunctive normal form (DNF) formula is defined analogously except that the symbols $\vee$ and $\wedge$ are interchanged. Every formula can be transformed to one in CNF or DNF. In general, this transformation will take exponential time. Let K be an algebraic number field. We showed that there is a polynomial time algorithm to decide whether or not a given $\exists\forall$ sentence in CNF or DNF is true in K [32]. In this paper, we will show that for every algebraic integer ring I, the decision problem of $\exists\forall$ sentences in CNF or DNF true in I is NP-complete. It follows that the similar problem for $\forall\exists$ sentences is co-NP-complete. We also will prove that there is a polynomial time algorithm to decide whether or not a given $\forall\exists$ sentence in CNF or DNF is true in every algebraic number ring. We then will prove that some related decision problems are in P.

Now we introduce our decision problems in the format used by Garey and Johnson [7]. We would like to emphasize the fact that if the decision problem is asked over a specific algebraic integer ring (AIR) I, then we have to include a monic, irreducible polynomial $g(t)\in\mathbf{Z}[t]$ in the instance. The ring I is the ring of integers of the algebraic number field $\mathbf{Q}[t]/g(t)$. Also in this case we extend the language L to include all the constants of I. This means that the formula may contain the elements of I, not just the elements of $\mathbf{Z}$.

$\forall^n\exists$ EQUATION OVER AIR

Instance: *Polynomial* $f(x_1,\cdots,x_n,y)$ *over* I *and polynomial* $g(t)$.
Question: *Is* $\forall x_1\cdots\forall x_n\exists y f(x_1,\cdots,x_n,y)=0$ *true in* I?

$\exists^n\forall$ INEQUATION OVER AIR

Instance: *Polynomial* $f(x_1,\cdots,x_n,y)$ *over* I *and polynomial* $g(t)$.
Question: *Is* $\exists x_1\cdots\exists x_n\forall y f(x_1,\cdots,x_n,y)\neq 0$ *true in* I?

$\forall\exists$ ($\exists\forall$) SENTENCE OVER AIR

Instance: CNF *or* DNF *formula* $\varphi(x, y)$ *containing no other free variables except* x, y, *and polynomial* $g(t)$.

Question: *Is* $\forall x \exists y\, \varphi(x, y)$ $(\exists x \forall y\, \varphi(x, y))$ *true in* I?

$\forall^n\exists$ EQUATION OVER ALL AIR

Instance: *Polynomial* $f(x_1, \cdots, x_n, y)$ *over* $\mathbf{Z}$.

Question: *Is* $\forall x_1 \cdots \forall x_n \exists y f(x_1, \cdots, x_n, y) = 0$ *true in every algebraic integer ring?*

$\forall\exists$ SENTENCE OVER ALL AIR (CIR, RIR, BIR)

Instance: CNF *or* DNF *formula* $\varphi(x, y)$ *containing no other free variables except* x *and* y.

Question: *Is* $\forall x \exists y\, \varphi(x, y)$ *true in every algebraic (cyclic, radical, abelian) integer ring?*

Notice that $\forall^n\exists$ EQUATION OVER AIR is the complement of $\exists^n\forall$ INEQUATION OVER AIR. Also, because the negation of a CNF formula is a DNF formula and vice versa, $\forall\exists$ SENTENCE OVER AIR is the complement of $\exists\forall$ SENTENCE OVER AIR. With the exception of $\forall^n\exists$ EQUATION OVER AIR and thus $\exists^n\forall$ INEQUATION OVER AIR, all other problems have not been previously shown to be decidable. The decidability of $\forall\exists$ sentences true in $\mathbf{Z}$ or $\mathbf{Q}$ is proved in [28].

The main results of this paper are as follows:

THEOREM 3.2. $\exists^n\forall$ INEQUATION OVER AIR *is* NP-*complete*.

COROLLARY 3.3. $\forall^n\exists$ EQUATION OVER AIR *is co*-NP-*complete*.

THEOREM 3.6. $\exists\forall$ SENTENCE OVER AIR *is* NP-*complete*.

COROLLARY 3.7. $\forall\exists$ SENTENCE OVER AIR *is co*-NP-*complete*.

THEOREM 4.3. $\forall^n\exists$ EQUATION OVER ALL AIR *is in* P.

THEOREM 4.5. $\forall\exists$ SENTENCE OVER ALL AIR *is in* P.

COROLLARY 4.6–4.8. $\forall\exists$ SENTENCE OVER ALL RIR (CIR, BIR) *is in* P.

This paper is a continuation of our study on arithmetical sentences [27]–[32].

1. Preliminaries. An element α is *algebraic* if and only if α satisfies an equation with coefficients in the rational number field $\mathbf{Q}$. An extension field F is algebraic over $\mathbf{Q}$ if and only if every element is algebraic. It is well known that every finite extension field is algebraic; the finite extensions of $\mathbf{Q}$ are called *algebraic number fields.* An extension field F of $\mathbf{Q}$ is a *cyclic (abelian) extension field* if and only if F is algebraic and Galois over $\mathbf{Q}$ and the Galois group of F over $\mathbf{Q}$ is a cyclic (abelian) group. An extension field F of $\mathbf{Q}$ is a *radical extension field* if and only if $F = \mathbf{Q}(u_1, \cdots, u_n)$, some power of u_1 lies in $\mathbf{Q}$ and for each $i \geqq 2$, some power of u_i lines in $\mathbf{Q}(u_1, \cdots, u_{i-1})$.

An algebraic number α is an *algebraic integer* if and only if it is a root of a monic polynomial over $\mathbf{Z}$. The set of algebraic integers of an algebraic number field forms a ring; it is called an *algebraic integer ring.* For the remainder of this paper we shall write K to denote an algebraic number field and I to denote the ring of integers of K. If K is a cyclic, abelian, or radical extension field, then we call I a *cyclic, abelian,* or *radical integer ring*, respectively.

Every algebraic number field K can be given as the field of rational number $\mathbf{Q}$ extended by a root α of a prescribed minimal polynomial $g(t) \in \mathbf{Z}[t]$ with the leading coefficient equal to 1; i.e., $K \simeq \mathbf{Q}(\alpha) \simeq \mathbf{Q}[t]/g(t)$. In our algorithms, we will work with the number field in its formulation as $\mathbf{Q}[t]/g(t)$, although certain of our proofs will be in terms of K or $\mathbf{Q}(\alpha)$.

This is the traditional representation of algebraic numbers. We can also represent algebraic numbers by suitable rational approximations. By the work of Kannan, Lenstra, and Lovász [10], the computations in either representation can be changed to computations in the other without the loss of efficiency. All the problems discussed in this paper, however, are still in the same computational complexity class no matter which representation is used.

Let β be an algebraic integer of K, then $\mathbf{Z}[\beta]$ is a subring of I. In general, these two rings are not equal. However, every algebraic integer ring I has an *integral basis* [19]. That is, there exist elements $\omega_1, \omega_2, \cdots, \omega_m$ of I such that every element α of I is of the form $\alpha = a_1\omega_1 + \cdots + a_m\omega_m$, where $a_1, \cdots, a_m$ are elements of $\mathbf{Z}$. Notice that by the construction of an integral basis $\omega_1, \cdots, \omega_m$ of I in [19], we may take $\omega_1 = 1$.

The notations about the computational complexity in this paper are taken from Garey and Johnson [7]. P is the class of sets S of strings such that there exists a deterministic Turing machine M and a polynomial f such that M accepts S and for all inputs x to M, M halts within $f(|x|)$ steps (where $|x|$ is the length of the input of x). NP is the class of sets S of strings such that there exists a nondeterministic Turing machine M and a polynomial f such that M accepts S and for all inputs x to M, one computation path of M halts within $f(|x|)$ steps. Also, co-NP is the class of sets S such that the complement of S is in NP. Let A and B be sets of strings. A is polynomial time reducible to B if there is a function F computable in polynomial time such that for all x, x belongs to A if and only if $F(x)$ belongs to B. A set B is NP-complete if it is in NP and for all A in NP, A is polynomial time reducible to B. A set B is co-NP-complete if the complement of B is NP-complete. In measuring computational complexity of arithmetical sentences, the polynomials are input in *nonsparse form*, i.e., if polynomial f contains the monomial $ax_1^{i_1} \cdots x_n^{i_n}$ with $a \neq 0$ and the monomial $bx_1^{j_1} \cdots x_n^{j_n}$ with $j_1 \leqq i_1, \cdots, j_n \leqq i_n$, then b must be input even if $b = 0$.

The main tool we will use in this paper is Theorem A [27, Prop. 1.1] below. Before we state the theorem, we define a term used in the theorem: An *arithmetic progression P in I* is a set $\{a + bi \mid i \in \mathbf{Z}\}$, where a, b are elements of I and $b \neq 0$.

THEOREM A. *Let $g \in K[x_1, \cdots, x_n, y]$. If for every finite sequence of arithmetic progressions $P_1, \cdots, P_n$ in I there exists integers x_i' of P_i $(1 \leqq i \leqq n)$ and an integer y' of I, such that $g(x_1', \cdots, x_n', y') = 0$, then*

(a) *$g(x_1, \cdots, x_n, y) = g_0(x_1, \cdots, x_n, y) \prod_{j=1}^{S} (y - f_j(x_1, \cdots, x_n))$, $s \neq 0$, where $f_j(x_1, \cdots, x_n) \in K[x_1, \cdots, x_n]$, $g_0(x_1, \cdots, x_n, y) \in K[x_1, \cdots, x_n, y]$ and $g_0(x_1, \cdots, x_n, y)$ has no factor of the form $y - f(x_1, \cdots, x_n)$,*

(b) *for any integers $x_1', \cdots, x_n'$ of I there exists a $j (1 \leqq j \leqq s)$ such that $f_j(x_1', \cdots, x_n') \in I$.*

Theorem A is deduced from Schinzel's theorem [25, Thm. 36] on diophantine equations with parameters. See [24], [25] for further discussion of the origins of Theorem A.

2. Basic algorithms. In this section we demonstrate that some basic operations and decision problems over an algebraic integer ring can be done in polynomial time. All of these results will be needed in later sections. We assign each algorithm a name, and hereafter will call the algorithms by their names.

LEMMA 2.1 (INTEGER). *Let K be an algebraic number field and β be a number of K, there is a polynomial time algorithm to decide whether β is an algebraic integer or not.*

Proof. Let $f(x)$ be the field equation of β over $\mathbf{Q}$ and $h(x)$ be the monic minimal polynomial of β over $\mathbf{Q}$. There exists a positive integer k such that $f(x) = h(x)^k$

[19, p. 11]. Hence the polynomial $f(x)$ must be a monic polynomial over $\mathbf{Q}$. For every algebraic number β there is a polynomial time algorithm to get the field equation of β over $\mathbf{Q}$ [19, p. 16–17]. According to the Gauss lemma, $f(x)$ is a polynomial over $\mathbf{Z}$ if and only if $h(x)$ is a polynomial over $\mathbf{Z}$. Hence, if $f(x)$ is a polynomial over $\mathbf{Z}$ then β is an algebraic integer; otherwise β is not an algebraic integer. □

Remark. If we know the degree, height, and sufficient rational approximation of an algebraic number, then we can find its minimal polynomial in polynomial time [10]. Therefore we can decide whether or not it is an algebraic integer in polynomial time. Let $\beta \in K = \mathbf{Q}(\alpha)$, then the degree of β is less than or equal to the degree of α. Also we can find an upper bound of the height of β if $f(\beta)=0$ for an $f(x)$ in $\mathbf{Z}[x]$ [20]. The algebraic numbers appearing in the algorithms of this paper are the roots of polynomial equations with known height. Thus all the analyses of the complexity of algorithms are still valid if we represent the algebraic numbers by suitable rational approximations.

Let a, b be two elements of I; a is *divisible* by b if and only if there exists an element c of I such that $a = b \cdot c$. From Lemma 2.1, we obtain the following corollary.

COROLLARY 2.2 (DIVISIBILITY). *Let a, b be two elements of algebraic integer ring I. There is a polynomial time algorithm to decide whether or not a is divisible by b.*

Proof. Let $K = \mathbf{Q}[\alpha] = \mathbf{Q}[t]/g(t)$ and $b = b_0 + b_1\alpha + \cdots + b_m\alpha^m = f(\alpha) \in K$. Apply the Euclidean algorithm to find the polynomials $u(x)$ and $v(x)$ of $\mathbf{Q}[x]$ such that $u(x)g(x) + v(x)f(x) = 1$. Then apply the algorithm INTEGER to check whether or not the element $a \cdot v(\alpha)$ is an integer. If $a \cdot v(\alpha)$ is an integer, then a is divisible by b; otherwise a is not divisible by b. □

LEMMA 2.3 (EQUATION). *There is a polynomial time algorithm for solving equations over I.*

Proof. Let $f(x) = 0$ be an equation over I. It is well known [9, 11, or 15] that there is a polynomial time algorithm to factor $f(x)$ over the quotient field K of I. If $f(x)$ does not have a linear factor $(x - a)$ then $f(x) = 0$ is not solvable in K. And thus, equation $f(x) = 0$ is not solvable in I. Suppose that $f(x) = 0$ is solvable in K. Let $\alpha_1, \cdots, \alpha_S$ be the roots of $f(x) = 0$ in K. We apply the algorithm INTEGER to decide whether or not α_i is an integer for every $i \leqq s$. If none of these α_i's is an integer, then $f(x) = 0$ is not solvable in I. If there exists an α_i such that α_i is an integer, then $f(x) = 0$ is solvable in I. We also obtain all the roots of $f(x) = 0$ in I. □

We call $\exists x\, \varphi(x)$ (or $\forall x\, \varphi(x)$) an $\exists$ sentence (or $\forall$ sentence) if and only if $\varphi(x)$ is a formula containing no quantifiers and no other free variables except x.

LEMMA 2.4 ($\exists$ SENTENCE). *There is a polynomial time algorithm to decide whether or not a* CNF *or* DNF $\exists$ *sentence is true in I.*

Proof. Let $\exists x\, \varphi(x)$ be an $\exists$ sentence in CNF i.e., $\varphi(x) \Leftrightarrow \bigwedge_{i=1}^{S} [\bigvee_{j=1}^{m_i} f_{i,j}(x) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x) \neq 0]$, where $f_{i,j}(x)$ and $g_{i,k}(x)$ are polynomials over I. If for every i there is a k such that $g_{i,k}(x) \not\equiv 0$, then $\exists x \varphi(x)$ is true. Now suppose that there is an i such that $g_{i,k}(x) \equiv 0$ for every $k \leqq n_i$. Without loss of generality, we may write that

$$\varphi(x) \Leftrightarrow \bigwedge_{i=1}^{p} \left[\bigvee_{j=1}^{m_i} f_{i,j}(x) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x) \neq 0 \right] \wedge \left[\bigvee_{l=1}^{t} f_l(x) = 0 \right].$$

We apply the algorithm EQUATION to solve the equation $f_l(x) = 0$ for every $l \leqq t$ in I. Let A be the set of all these solutions. If there is an a of A such that $\bigwedge_{i=1}^{p} [\bigvee_{j=1}^{m_i} f_{i,j}(a) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(a) \neq 0]$ is true, then $\exists x\, \varphi(x)$ is true in I. Otherwise, $\exists x\, \varphi(x)$ is false in I.

Now let $\varphi(x)$ be a formula in DNF, i.e., $\varphi(x) \Leftrightarrow \bigvee_{i=1}^{S} [\bigwedge_{j=1}^{m_i} f_{i,j}(x) = 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(x) \neq 0]$, where $f_{i,j}(x)$ and $g_{i,k}(x)$ are polynomials over I. We apply the algorithm EQUATION to solve the equations $f_{i,j}(x) = 0$, $g_{i,k}(x) = 0$ in I for every i, j,

and k. (If $m_i > 1$ for some i, it will be more efficient to find the greatest common divisor, GCD, of polynomials $f_{i,j}$ for $j \leqq m_i$, and then find the roots of GCD.) If there exist an i and an a such that $\bigwedge_{j=1}^{m_i} f_{i,j}(a) = 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(a) \neq 0$, then $\exists x\, \varphi(x)$ is true. Otherwise $\exists x\, \varphi(x)$ is false. □

We also obtain the following corollary.

COROLLARY 2.5 (∀ SENTENCE). *There is a polynomial time algorithm to decide whether or not a* CNF *or* DNF ∀ *sentence is true in I.*

Proof. The negation of an ∀ sentence is an ∃ sentence. Also the negation of a CNF formula is a DNF formula and vice versa. The decision problem ∀ SENTENCE is the complement of ∃ SENTENCE. The class P is closed under complement [7]. This proves our corollary. □

3. Sentences over an algebraic integer ring. In this section we prove the results about sentences over an algebraic integer ring. We first introduce a decision problem called SIMULTANEOUS INCONGRUENCES, which is needed to prove our NP-completeness results. This problem is a generalization of the problem under the same title in [7] to algebraic integer rings. Now we give a necessary definition. Let a, b, and c be elements of an algebraic integer ring I. We say that a is congruent to b modulo c if there is an integer d of I such that $a - b = c \cdot d$, and we write $a \equiv b \pmod{c}$.

SIMULTANEOUS INCONGRUENCES

Instance: *Collection* $\{(a_1, b_1), \cdots, (a_n, b_n)\}$ *of ordered pairs of integers in I.*

Question: *Is there an integer x of I such that, for* $1 \leqq i \leqq n$, $x \not\equiv a_i \pmod{b_i}$ *in I*?

With the algorithm DIVISIBILITY, it is easy to see that SIMULTANEOUS INCONGRUENCES is in NP. We will prove this problem is NP-complete. In the proof of completeness we use the fact that every algebraic integer ring has an integral basis over **Z**. There are algorithms to obtain an integral basis [19], but we do not know whether there is a polynomial time algorithm for doing this or not. This problem is deterministic polynomial time equivalent to the problem of computing the greatest square divisor of a rational integer [17]. Thus this problem is in NP and might not be easy. Fortunately, we need to obtain only one such basis. Our transformation does not depend on how we obtain such an integral basis.

LEMMA 3.1. SIMULTANEOUS INCONGRUENCES *is* NP-*complete.*

Proof. We prove the completeness by following the original proof in [26]. The only difference is "encoding."

Let A be a Boolean formula in CNF with three literals per conjunct. Let C_k be the set of literals in the kth conjunct, $1 \leqq k \leqq p$. Say that A has n distinct variables, so that an assignment to the variables can be represented as a binary vector of length n. Let $p_1, p_2, \cdots, p_n$ be the first n primes in **N**. We say z of I encodes an assignment if z is congruent to 0 or 1 modulo p_i for $1 \leqq i \leqq n$. We say z satisfies A if the assignment $[z \bmod p_1, z \bmod p_2, \cdots, z \bmod p_n]$ satisfies A.

Let $\{\omega_1 = 1, \omega_2, \cdots, \omega_m\}$ be an integral basis of I. The number z of I encodes an assignment if and only if for every i, $1 \leqq i \leqq n$,

$$z \not\equiv r_1 + r_2\omega_2 + \cdots + r_m\omega_m \pmod{p_i} \quad \text{for all}$$

$$2 \leqq r_1 \leqq p_i \quad \text{and} \quad 0 \leqq r_j \leqq p_i \quad \text{for} \quad 2 \leqq j \leqq m.$$

For each conjunct C_k, we construct a system of three congruence equations such that z_k does not assign the value 1 to any literal in C_k. For example, if $C_k = \{x_r, \neg x_s, x_t\}$ for $1 \leqq r, s, t \leqq n$ and r, s, t distinct, then let $z_k \equiv 0 \pmod{p_r}$, $z_k \equiv 1 \pmod{p_s}$, $z_k \equiv 0$

(mod p_t). Now we can see that A is satisfiable if and only if there exists a z of I; z encodes an assignment and $z \not\equiv z_k \pmod{p_r p_s p_t}$ for every $1 \leqq k \leqq p$. We can see that our transformation is in polynomial time. Our conclusion follows from the well-known fact that SAT is NP-complete [2] or [7]. □

Now we prove our first main result. The decidability of this problem was shown in [27]. The main difference between these two algorithms is that this time we apply Lenstra's polynomial time algorithm [15] instead of the classical Kronecker's algorithm to factor the polynomials. Then we can give the upper and lower bound of its computational complexity.

THEOREM 3.2. $\exists^n\forall$ INEQUATION OVER AIR *for every* $n \geqq 1$ *is* NP-*complete.*

Proof. We first prove that this problem is in NP. Let $f(x_1, \cdots, x_n, y)$ be a polynomial over I.

(S_1) We factor $f(x_1, \cdots, x_n, y)$ over the algebraic number field K. This can be done in polynomial time [15]. If there is no $F(x_1, \cdots, x_n)$ over K such that $y - F(x_1, \cdots, x_n)$ is a factor of $f(x_1, \cdots, x_n, y)$, then by Theorem A, there exist $x'_1, \cdots, x'_n$ of I such that for every y' of I, $f(x'_1, \cdots, x'_n, y') \neq 0$. Now suppose that $f(x_1, \cdots, x_n, y)$ has factors of the form $y - F(x_1, \cdots, x_n)$ and let S be the set of these polynomials $F(x_1, \cdots, x_n)$.

(S_2) Let *discr*(g) denote the *discriminate* of $g(t)$ where $K \simeq \mathbf{Q}[t]/g(t) \simeq \mathbf{Q}[\alpha]$ and let d be an element of $\mathbf{Z}$ such that $d \cdot f$ is a polynomial with coefficients in $\mathbf{Z}[\alpha]$. It is well known [33] that if we take $D = d \cdot |discr(g)|$, then all monic (with respect to y) factors of $f(x_1, \cdots, x_n, y)$ are in $(1/D)\mathbf{Z}[\alpha][x_1, \cdots, x_n, y]$. (In [33] only a univariate lemma is proved. The multivariate version can be deduced by a Kronecker's substitution.) Let m be the degree of $g(t)$ and $\{\omega_1, \cdots, \omega_m\}$ be an integral basis of I over $\mathbf{Z}$. The representatives of the residue classes of the ideal (D) over I are in the form $r_1\omega_1 + \cdots + r_m\omega_m$, where r_l are elements of $\mathbf{Z}$ and $0 \leqq r_l < D$ [19, p. 49]. Notice that the input length of every representative is polynomially bounded by the input length of $f(x_1, \cdots, x_n, y)$ and I. Let b_i, c_i be elements of I and $c_i \equiv b_i \pmod D$ for every $i \leqq n$. We obtain that for every $F(x_1, \cdots, x_n)$ of S, if $F(b_1, \cdots, b_n)$ is an element of I, then $F(c_1, \cdots, c_n)$ is an element of I. Hence to see whether or not there exist integers $x'_1, \cdots x'_n$ of I such that $F(x'_1, \cdots, x'_n)$ is not an integer, we need only to apply the algorithm INTEGER to check the number $F(a_1, \cdots, a_n)$ for all the representatives $a_1, \cdots, a_n$ of the residue classes of I modulo (D). If for all the representatives $a_1, \cdots, a_n$ of the residue classes, there exists an $F(x_1, \cdots, x_n)$ of S, such that $F(a_1, \cdots, a_n)$ is an element of I, then it is easy to see that $\forall x_1 \cdots \forall x_n \exists y\, f(x_1, \cdots, x_n, y) = 0$ is true in I. If there exist representatives $x'_1, \cdots, x'_n$, such that $F(x'_1, \cdots, x'_n)$ is not an integer for every $F(x_1, \cdots, x_n)$ of S, then $\exists x_1 \cdots \exists x_n\, \forall y\, f(x_1, \cdots, x_n, y) \neq 0$ is true in I by Theorem A. This can be done in "*nondeterministic*" polynomial time. Therefore $\exists^n\forall$ INEQUATION OVER AIR is in NP for $n \geqq 1$.

Now we need to prove the completeness. Let $\{(a_1, b_1), \cdots, (a_n, b_n)\}$ be the instance of SIMULTANEOUS INCONGRUENCES. Let $f(x, y)$ be the polynomial $\prod_{i=1}^n (x - b_i y - a_i)$. It is easy to see that $\exists x \forall y f(x, y) \neq 0$ is true in I if and only if the answer to SIMULTANEOUS INCONGRUENCES is positive. Therefore $\exists\forall$ INEQUATION OVER AIR is NP-complete. Hence $\exists^n\forall$ INEQUATION OVER AIR for every $n \geqq 1$ is NP-complete. □

By the definition of co-NP-completeness we obtain the following corollary.

COROLLARY 3.3. $\forall^n\exists$ EQUATION OVER AIR *is co*-NP-*complete for* $n \geqq 1$.

It is well known that if a decision problem $\mathbf{X}$ is co-NP-complete, then it is not in NP unless NP = co-NP [7]. Therefore, $\forall^n\exists$ EQUATION OVER AIR, for $n \geqq 1$, is not

likely to be in NP. Also if $n=0$, then this is simply the decision problem of solvability of equations in I. By the algorithm EQUATION this problem is in P.

Remark. The parameters, i.e., universal quantifiers, of $\forall^n\exists$ EQUATION OVER AIR can be restricted to **N** or **Z** and we will still obtain the same results. This means that the following decision problems are co-NP-complete. Given an arbitrary polynomial $f(x_1, \cdots, x_n, y)$ over I, whether or not for any integers $x_1', \cdots, x_n'$ of **N** (or **Z**), the equation $f(x_1', \cdots, x_n', y)=0$ is solvable in I. Notice that if we also ask the solvability in **N** or **Z** instead of I, we then have $\forall^n\exists$ EQUATION OVER **N** or $\forall^n\exists$ EQUATION OVER **Z** respectively. The decision problem of $\forall^n\exists$ EQUATION OVER **N** is undecidable [27], whereas the decision problem of $\forall^n\exists$ EQUATION OVER **Z** is co-NP-complete [29], a special case of Corollary 3.3.

To prove the above new NP-complete results we need to modify our Theorem A. In the following theorem we both assume less and assert less than in Theorem A. The proofs of these two theorems are the same, hence we will not repeat ourselves.

THEOREM B. *Let $g\in K[x_1, \cdots, x_n, y]$. If for every finite sequence of arithmetic progressions $P_1, \cdots, P_n$ in* **Z** *there exist integers x_i' of P_i $(1\leqq i\leqq n)$ and an integer y' of I, such that $g(x_1', \cdots, x_n', y')=0$, then*

(a) $g(x_1, \cdots, x_n, y)=g_0(x_1, \cdots, x_n, y)\prod_{j=1}^{S}(y-f_j(x_1, \cdots, x_n))$, $s\neq 0$, *where* $f_j(x_1, \cdots, x_n)\in K[x_1, \cdots, x_n]$, $g_0(x_1, \cdots, x_n, y)\in K[x_1, \cdots, x_n, y]$ *and* $g_0(x_1, \cdots, x_n, y)$ *has no factor of the form* $y-f(x_1, \cdots, x_n)$,

(b) *for any integers $x_1', \cdots, x_n'$ of* **Z** *there exists a j $(1\leqq j\leqq s)$ such that $f_j(x_1', \cdots, x_n')\in I$.*

Then with the same arguments as Theorem 3.2 we can prove the remark above.

Now we apply Theorem A to obtain a technical result.

PROPOSITION 3.4. *Let K be an algebraic number field, I be the ring of integers of K, and $\varphi(x, y)$ be a formula of the form $\bigvee_{i=1}^{S}[\bigwedge_{j=1}^{m_i} f_{i,j}(x, y)=0]$, where $f_{i,j}(x, y)$ are polynomials over K. If for every arithmetic progression P in I there exist an x' of P and an integer y' of I such that $\varphi(x', y')$ is true, then*

(a) *there exist an i $(l\leqq i\leqq s)$ and a polynomial $F(x)$ over K such that $y-F(x)$ is a common factor of the polynomials $f_{i,j}(x, y)$ $(1\leqq j\leqq m_i)$.*

(b) *for every element x' of I, there exists an $F(x)$ satisfying the conditions of* (a) *such that $F(x')$ is an element of I.*

Proof. For every x' and y' of I, if $\varphi(x', y')$ is true, then $\bigvee_{i=1}^{S} f_{i,1}(x', y')=0$, or $\prod_{i=1}^{S} f_{i,1}(x', y')=0$ is true. Hence for every arithmetic progression P in I there exist an x' of P and an integer y' of I such that $\prod_{i=1} f_{i,1}(x', y')=0$. From Theorem A there exists a polynomial $F_1(x)$ over K such that $y-F_1(x)$ is a factor of the polynomial $\prod_i F_{i,1}(x, y)$. Then $y-F_1(x)$ must be an irreducible factor of the polynomial $f_{t,1}(x, y)$ for a $t\leqq s$. Now suppose that $y-F_1(x)$ is not a factor of one of the polynomials $f_{t,j}(x, y)$ for $1<j\leqq m_t$. Let B be the set of x's of the common roots of the equations $y-F_1(x)=0$ and $f_{t,j}(x, y)=0$; B is a finite set. If a set T intersects every arithmetic progression in I, then T intersects every arithmetic progression with infinitely many terms. Let R be the set of x' of I such that there exists an integer y' of I, and $\varphi(x', y')$ is true. By our assumptions, R intersects every arithmetic progression in I, in fact, with infinitely many terms. Therefore $R-B$ intersects every arithmetic progression in I. For every x' of $R-B$ there exists an integer y' of I such that $g(x', y')=0$, where $g(x, y)=\prod_i f_{i,1}(x, y)/(y-F_1(x))$. By Theorem A again, the polynomial $g(x, y)$ must have a factor $y-F_2(x)$, where $F_2(x)$ is a polynomial over K. The polynomial $y-F_2(x)$ might not satisfy the conditions of (a). Then with the same arguments, the polynomial $\prod_i f_{i,1}(x, y)/(y-F_1(x))\cdot(y-F_2(x))$ must have a factor $y-F_3(x)$, where $F_3(x)$ is a polynomial over K. Since the degree of y in the polynomial $\prod_i f_{i,1}(x, y)$ is finite, there

must exist a polynomial $y-F(x)$ satisfying the conditions of (a). Now we need to prove part (b). Let S be the set of all the polynomials $F(x)$ over K such that $y-F(x)$ is a common factor of the polynomials $f_{i,j}(x, y)$ for a fixed i $(1 \leqq i \leqq s)$ and for all j $(1 \leqq j \leqq m_i)$. For every $F(x)$ there exists a rational integer b, $b \neq 0$, such that $b \cdot F(x)$ is a polynomial over I. Such a number b exists because for every algebraic number β of K, there is a rational integer r, $r \neq 0$, such that $r \cdot \beta$ is an algebraic integer. Thus there is a rational integer a, $a \neq 0$, such that $a \cdot F(x)$ is a polynomial over I for every $F(x)$ of S. Suppose that there exists an integer t of I such that for every $F(x)$ of S, $F(t)$ is not an integer of I. This is equivalent to $a \cdot F(t) \not\equiv 0 \pmod{a}$ for every $F(x)$ of S. If we let $s \equiv t \pmod{a}$, it follows that $a \cdot F(s) \equiv a \cdot F(t) \not\equiv 0 \pmod{a}$, hence $F(ac+t)$ is not an integer of I for every $F(x)$ of S and every c of I. Now let $F_{i,j}(x, y) = f_{i,j}(x, y)/\prod_k (y - F_k(x))$, where polynomials $y - F_k(x)$ are common factors of $f_{i,j}(x, y)$ for all j $(1 \leqq j \leqq m_i)$. Hence $F_{i,j}(x, y)$ are polynomials over K and for every i the polynomials $F_{i,j}(x, y)$ $(1 \leqq j \leqq m_i)$ have no common factors of the form $y - F(x)$. It follows that for every i the polynomials $F_{i,j}(ax+t, y)$ $(1 \leqq j \leqq m_i)$ have no common factors of the form $y - F(x)$. By part (a) there exist integers d and e of I such that if x' is in the arithmetic progression $\{x = di + e \mid i \in \mathbf{Z}\}$, then for every y' of I, $\bigvee_{i=1}^{S} [\bigwedge_{j=1}^{m_i} F_{i,j}(ax'+t, y') = 0]$ is not true. The arithmetic progression $P = \{a(di+e)+t \mid i \in \mathbf{Z}\}$, where a, d, e, and t are integers as above, has the property that for any x' of P and any y' of I the sentence $\bigvee_{i=1}^{S} [\bigwedge_j f_{i,j}(x', y') = 0]$ is not true. This contradicts our assumptions. Therefore for every x' of I there exists an $F(x)$ such that $F(x')$ is an element of I. $\square$

Before we prove another main theorem of this paper, ∃∀ SENTENCE OVER AIR is NP-complete, we prove that for a special form of ∃∀ sentences the decision problem is in P. We need this result in the proof of ∀∃ SENTENCE OVER AIR is NP-complete. Since these two cases are in two different computational complexity classes if $P \neq NP$, this special case may have independent interest.

PROPOSITION 3.5. *Let I be an algebraic integer ring and $\exists x \forall y\, \varphi(x, y)$ be a sentence of the form $\exists x \forall y \bigwedge_{i=1}^{S} [\bigvee_{j=1}^{m_i} f_{i,j}(x, y) \neq 0 \vee \bigvee_{k=1}^{m_i} g_{i,j}(x, y) = 0]$, where $f_{i,j}(x, y)$, $g_{i,k}(x, y)$ are polynomials over I and there is a t $(1 \leqq t \leqq s)$ such that $f_{t,j}(x, y) \equiv 0$ for every j $(1 \leqq j \leqq m_t)$ and $g_{t,k}(x, y) \not\equiv 0$ for every k $(1 \leqq k \leqq n_t)$. Then there is a polynomial time algorithm to decide whether or not $\exists x \forall y\, \varphi(x, y)$ is true in I.*

Proof. We assume that $f_{t,j}(x, y) \equiv 0$ for every $1 \leqq j \leqq m_t$ and $g_{t,k}(x, y) \not\equiv 0$ for every k, $1 \leqq k \leqq n_t$. In order to simplify the notations, we omit the index t for the remainder of this proof. Let $g_k(x, y) = \sum_p h_{k,p}(x) \cdot y^p$, where $h_{k,p}(x)$ are polynomials over I and for every k there is a p such that $h_{k,p}(x) \not\equiv 0$. We apply the algorithm EQUATION to solve all the equations $h_{k,p}(x) = 0$ in I simultaneously. Let A be the subset of I such that if a is an element of A, then there is a k such that $h_{k,p}(a) = 0$ for every p, i.e., $g_k(a, y) \equiv 0$. For every k, $g_k(x, y) \not\equiv 0$, A is a finite set. If x' is an element of I, such that $\forall y \bigwedge_{i=1}^{S} [\bigvee_{j=1} f_{i,j}(x', y) \neq 0 \vee \bigvee_{k=1} g_{i,k}(x', y) = 0]$ is true in I, then x' must be an element of A. If none of the polynomials $g_k(x', y)$ is identically equal to zero, then this sentence cannot be true in I. Now suppose that the set A is nonempty. The number of elements in A must be less than the sum of the degrees of x in $g_k(x, y)$ for $1 \leqq k \leqq n$. Also the input length of elements of A are polynomially bounded by the input length of the formula $\varphi(x, y)$. We apply the algorithm ∀ SENTENCE to check the sentence $\forall y\, \varphi(x', y)$ for each x' of A to see whether or not it is true in I. If there is an x' of A such that $\forall y\, \varphi(x', y)$ is true in I, then $\exists x \forall y\, \varphi(x, y)$ is true in I. Otherwise, it is false in I. $\square$

Now we prove our second main result.

THEOREM 3.6. ∃∀ SENTENCE OVER AIR *is* NP-*complete.*

Proof. We first show that this problem is in NP. Let $\varphi(x, y)$ be a formula in CNF containing no other free variables.

(S_1) We may write that $\varphi(x, y)=\bigwedge_{i=1}^{s} \varphi_i(x, y)$, where $\varphi_i(x, y)=\bigvee_{j=1}^{m_i} f_{i,j}(x, y)\neq 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y)=0$, $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are polynomials over I. If there is a t, $1\leqq t\leqq s$, such that $g_{t,k}(x, y)\equiv 0$ for a k ($1\leqq k\leqq n_t$), then we may omit the term $\varphi_t(x, y)$ from the formula $\varphi(x, y)$ to obtain the formula $\varphi'(x, y)$, and $\exists x\forall y\, \varphi(x, y)$ is true if and only if $\exists x\forall y\, \varphi'(x, y)$ is true. Therefore, without loss of generality, we may assume that $g_{i,k}(x, y)\not\equiv 0$ for every i and k. Now suppose that there is a t ($1\leqq t\leqq s$) such that $f_{t,j}(y)\equiv 0$ for every j, $1\leqq j\leqq m_t$, then by Proposition 3.5 there is a polynomial time algorithm to decide whether or not $\exists x\forall\, \varphi(x, y)$ is true in I. Now we may assume that $f_{i,j}(x, y)\not\equiv 0$ for every i and j.

(S_2) We factor all the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ over the quotient field K of I. Factorization of polynomials over an algebraic number field is in polynomial time [15]. For every i, we may omit from $f_{i,j}(x, y)$ ($1\leqq j\leqq m_i$) any factors also occurring in one of the polynomials $g_{i,k}(x, y)$ ($1\leqq k\leqq n_i$). This will not affect the truth of the sentence $\exists x\forall y\, \varphi(x, y)$. Therefore we assume that for every i, j, and k the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime. If there is no $F(x)$ over K such that $y-F(x)$ is a common factor of the polynomials $f_{i,j}(x, y)$ ($1\leqq j\leqq m_i$), then by Proposition 3.4 the sentence $\forall x\exists y \bigvee_i [\bigwedge_j f_{i,j}(x, y)=0]$ is false in I. It follows that the sentence $\exists x\forall y\, \varphi(x, y)$ is true in I. Now we suppose that there exist an i and a polynomial $F(x)$ such that $y-F(x)$ is a common factor of the polynomials $f_{i,j}(x, y)$ ($1\leqq j\leqq m_i$). Notice that $y-F(x)$ is not a factor of any one of the polynomials $g_{i,k}(x, y)$ ($1\leqq k\leqq n_i$), since for every i, j and k, $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime. Let S be the set of these polynomials $F(x)$.

(S_3) Let $d_i\in \mathbf{Z}$ be such that $f_{i,1}(x, y)\in(1/d_i)\mathbf{Z}[\alpha][x, y]$, and let $discr(g)$ denote the discriminant of $g(t)$ where $K\simeq \mathbf{Q}[t]/g(t)$. It is well known [cf. 33] that if we take $D_i=d_i\cdot |discr(g)|$, then all monic (with respect to y) factors of $f_{i,1}(x, y)$ are in $(1/D_i)\mathbf{Z}[\alpha][x, y]$. Let D be the least common multiple of D_i, $1\leqq i\leqq s$. It follows that $D\cdot F(x)$ is a polynomial over I for every $F(x)$ of S. Let $\{\omega_1, \cdots, \omega_m\}$ be an integral basis of I over $\mathbf{Z}$. The representatives of the residue classes of the ideal (D) over I are in the form $r_1\omega_1+\cdots+r_m\omega_m$, where r_l are elements of $\mathbf{Z}$ and $0\leqq r_l<D$. For every $F(x)$ of S if $F(b)$ is an integer of I and $c\equiv b \pmod{D}$, then $F(c)$ is an integer of I. Hence to see whether or not there exists an integer x' of I such that $F(x')$ is not an integer, we only need to apply the algorithm INTEGER to check $F(a)$ for every $F(x)$ of S and every representative a of the residue classes of I modulo (D). If there is a representative a such that $F(a)$ is not an element of I for every $F(x)$ of S, then $\exists x\forall y\, \varphi(x, y)$ is true in I by part (b) of Proposition 3.4. This step can be done in "*nondeterminstic*" polynomial time. Now suppose that for every integer x' there is a polynomial $F(x)$ of S such that $F(x')$ is an element of I. In this case, if for every i the equation $g_{i,k}(x, F(x))=0$, where $y-F(x)$ is the common factor of $f_{i,j}(x, y)$ ($1\leqq j\leqq m_i$), has no root in I for every $1\leqq k\leqq n_i$, then $\forall x\exists y \bigvee_{i=1} [\bigwedge_j f_{i,j}(x, y)=0 \wedge \bigwedge_k g_{i,k}(x, y)\neq 0]$ is true in I. Hence $\exists x\forall y\, \varphi(x, y)$ is false in I. In order for $\forall y\, \varphi(x', y)$ to be true in I, x' must be a root of one of the equations $g_{i,k}(x, F(x))=0$.

(S_4) We apply the algorithm EQUATION to solve the equations $g_{i,k}(x, F(x))=0$ in I for every i and k, where $F(x)$ is the polynomial that $f_{i,j}(x, F(x))\equiv 0$ for every j. Let T be the set of all the roots of these equations. We then apply the algorithm $\forall$ SENTENCE to check the sentence $\forall y\, \varphi(x', y)$ for every x' of T. If there is an x' of T such that $\forall y\, \varphi(x', y)$ is true in I then, of course, $\exists x\forall y\, \varphi(x, y)$ is true in I. If for every x' of T, the sentence $\forall y\, \varphi(x', y)$ is false, then $\forall x\exists y \neg \varphi(x, y)$ is true in I, i.e., $\exists x\forall y\, \varphi(x, y)$ is false in I.

The steps (S_1), (S_2), and (S_4) can be done in polynomial time. The step (S_3) can be done in *nondeterministic* polynomial time. Hence $\exists\forall$ SENTENCE OVER AIR is in NP if the formula $\varphi(x, y)$ is in CNF.

Now let $\varphi(x, y)$ be a formula in DNF, i.e.,

$$\varphi(x, y) = \bigvee_{i=1}^{S} \left[\bigwedge_{j=1}^{m_i} f_{i,j}(x, y) \neq 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(x, y) = 0 \right]$$

where $f_{i,j}(x, y)$, $g_{i,k}(x, y)$ are polynomials over I.

(T_1) We apply the Euclidean algorithm to find the greatest common divisor (GCD) of $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ for every i, j, and k. We then omit the GCD from $g_{i,k}(x, y)$. Thus without loss of generality, we may assume that for every i the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$, for $1 \leqq j \leqq m_i$ and $1 \leqq k \leqq n_i$, are relatively prime.

(T_2) Suppose that for each i there is a k such that $g_{i,k}(x, y) \not\equiv 0$. Then for every i we need to find the number x' of I such that $g_{i,k}(x', y) \equiv 0$ for all $k \leqq n_i$. This means that for a polynomial $g(x, y) = h_m(x)y^m + \cdots + h_0(x)$ we need to find the common solutions of $\{h_m(x) = 0, \cdots, h_0(x) = 0\}$. Therefore we apply the Euclidean algorithm to find the GCD, $h(x)$, of the polynomials $h_m(x), \cdots, h_0(x)$, then apply the algorithm EQUATION to find the roots of $h(x) = 0$ in I. Let $A_{i,k}$ be the set of x' of I such that $g_{i,k}(x', y) \equiv 0$ if $x' \in A_{i,k}$ and $A = \bigcup_{i=1}^{S} \bigcap_{k=1}^{n_i} A_{i,k}$. We apply the algorithm $\forall$ SENTENCE to check the truth of the sentence $\forall y\, \varphi(x', y)$ for every x' of A. Since the number of elements of A is polynomially bounded, this can be done in polynomial time. If there is an x' of A such that $\forall y\, \varphi(x', y)$ is true, then $\exists x \forall y\, \varphi(x, y)$ is true. Otherwise, $\exists x \forall y\, \varphi(x, y)$ is false. Now we may assume that for some i, $g_{i,k}(x, y) \equiv 0$ for every $k \leqq n_i$. Since $a \neq 0 \wedge b \neq 0 \Leftrightarrow a \cdot b \neq 0$, we can combine several inequations into one and obtain that

$$\varphi(x, y) \Leftrightarrow \bigvee_{i=1}^{t} \left[\bigwedge_{j=1}^{m_i} f_{i,j}(x, y) \neq 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(x, y) = 0 \right] \vee \bigvee_{l=1}^{p} F_l(x, y) \neq 0,$$

where $f_{i,j}$, $g_{i,k}$, F_l are polynomials over I and $g_{i,k}(x, y) \not\equiv 0$. (In most cases combining inequations will increase the computation time in the following steps. However, by doing so our formulas and proofs are simpler.)

(T_3) We apply the Euclidean algorithm to find the GCD, $G(x, y)$, of polynomials $F_l(x, y)$ for $1 \leqq l \leqq p$, then factorize $G(x, y)$ over K by Lenstra's algorithm [15]. We may write that

$$G(x, y) = G_0(x, y) \prod_{\beta=1}^{q} (y - G_\beta(x)),$$

where $G_0(x, y)$ has no factor of the form $y - G(x)$. If $q = 0$, i.e., $G(x, y)$ has no factor of the form $y - G(x)$, then $\exists x \forall y \bigvee_{l=1}^{p} F_l(x, y) \neq 0$ is true in I by Proposition 3.4. Hence $\exists x \forall y\, \varphi(x, y)$ is true in I. Now we assume that $q \neq 0$.

(T_4) Let $H_{i,k,\beta}(x) = g_{i,k}(x, G_\beta(x))$, T_i be the set of polynomials $G_\beta(x)$ such that $H_{i,k,\beta}(x) \equiv 0$ for every $k \leqq n_i$ and $T = \bigcup_{i=1}^{t} T_i$. Let $G'(x, y) = G_0(x, y) \prod_{\delta=1}^{r} (y - G_\delta(x))$, where $G'(x, y)$ is the polynomial omitting from $G(x, y)$ any factor $y - G(x)$ if $G(x)$ is in T. We claim that if there is an x' of I such that $G_\delta(x')$ is not an element of I for every δ, $1 \leqq \delta \leqq r$, then $\exists x \forall y\, \varphi(x, y)$ is true in I. Suppose, on the contrary, that

$$\forall x \exists y \bigwedge_{i=1}^{t} \left[\bigvee_{j=1}^{m_i} f_{i,j}(x, y) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y) \neq 0 \right] \wedge \bigwedge_{l=1}^{p} F_l(x, y) = 0$$

is true in I. Let A be the set of elements of I such that if a is in A, then $G_\beta(a) \in I$ for

some $\beta \leqq q$, where $g_{i,k}(x, G_\beta(x)) \equiv 0$ for a fixed i and all $k \leqq n_i$, and

$$\bigwedge_{i=1}^{t} \left[\bigvee_{j=1}^{m_i} f_{i,j}(a, G_\beta(a)) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(a, G_\beta(a)) \neq 0 \right].$$

This implies that there is an i such that $\bigvee_{j=1}^{m_i} f_{i,j}(a, G_\beta(a)) = 0$. Since for each i, $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime, $f_{i,j}(x, G_\beta(x)) \not\equiv 0$. Hence A is finite. For every x of $I - A$

$$\exists y \bigwedge_{i=1}^{t} \left[\bigvee_{j=1}^{m_i} f_{i,j}(x, y) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y) \neq 0 \right] \wedge \bigwedge_{l=1}^{p} F'_l(x, y) = 0$$

is true in I, where $F'_l(x, y)$ is the polynomial omitting from $F_l(x, y)$ any factor $y - G(x)$ if $G(x)$ is in T. Then for every x of $I - A$, $\exists y \bigwedge_{l=1}^{p} F'_l(x, y) = 0$ is true. By Proposition 3.4, for every x' of I there exists a $G_\delta(x)$, $1 \leqq \delta \leqq r$, such that $G_\delta(x')$ is an element of I. This proves our claim. Now we check whether there is an a of I such that for every δ, $1 \leqq \delta \leqq r$, $G_\delta(a)$ is not an element of I. This can be done in "*nondeterministic*" polynomial time as we have shown in the step (S_3). If there is such an a of I, then $\exists x \forall y\, \varphi(x, y)$ is true in I by our claim. Now we assume that for every x of I there is a δ, $1 \leqq \delta \leqq r$, such that $G_\delta(x)$ is in I.

(T_5) We solve $H_{i,k,\delta}(x) = g_{i,k}(x, G_\delta(x)) = 0$ for every i, k, and δ. Let $A_{i,k,\delta}$ be the set of solutions for each equation. Let $A = \bigcup_{i=1}^{t} \bigcap_{k=1}^{n_i} \bigcup_{\delta=1}^{r} A_{i,k,\delta}$, A is still polynomially bounded by our original input. We check the sentence

$$\forall y \bigvee_{i=1}^{t} \left[\bigwedge_{j=1}^{m_i} f_{i,j}(a, y) \neq 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(a, y) = 0 \right] \vee \bigvee_{l=1}^{p} F_l(a, y) \neq 0$$

for every a of A by the algorithm $\forall$ SENTENCE. If there is an a of A that this sentence is true, then $\exists x \forall y\, \varphi(x, y)$ is true. Otherwise,

$$\forall x \exists y \bigwedge_{i=1}^{t} \left[\bigvee_{j=1}^{m_i} f_{i,j}(x, y) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y) \neq 0 \right] \wedge \bigwedge_{l=1}^{p} F_l(x, y) = 0$$

is true in I. This means that $\exists x \forall y\, \varphi(x, y)$ is false. Except step (T_4), which can be done in "*nondeterministic*" polynomial time, all other steps can be done in polynomial time. This completes our proof that $\exists\forall$ SENTENCE OVER AIR is in NP.

Now we prove the completeness. Notice that $\exists\forall$ INEQUATION OVER AIR is only a special case of $\exists\forall$ SENTENCE OVER AIR. Also $\exists\forall$ INEQUATION OVER AIR is NP-complete by Theorem 3.2. We obtain that $\exists\forall$ SENTENCE OVER AIR is NP-complete. □

We also have the following corollary.

COROLLARY 3.7. $\forall\exists$ SENTENCE OVER AIR *is co*-NP-*complete.*

Proof. This decision problem is the complement of $\exists\forall$ SENTENCE OVER AIR. □

4. Sentences over a class of rings. In this section we study the sentences true in some classes of rings of algebraic integers. We first give a necessary and sufficient condition for a diophantine equation with parameters to be solvable in a class of algebraic integer rings. With this condition we can prove that many decision problems are in P.

Let ζ_m be the primitive mth root of unity. The field $\mathbf{Q}(\zeta_m)$ is called the *cyclotomic field* of order m. Let $\varphi(m)$ be Euler's φ-function and $r = \varphi(m) - 1$. It is well known [19] that $1, \zeta_m, \cdots, \zeta_m^r$ form an integral basis of $\mathbf{Q}(\zeta_m)$. Let $\mathbf{Z}[\zeta_m]$ be the ring generated by ζ_m over $\mathbf{Z}$. Then $\mathbf{Z}[\zeta_m]$ is the ring of integers of the field $\mathbf{Q}(\zeta_m)$. We call $\mathbf{Z}[\zeta_m]$ the *cyclotomic integer ring* of order m. Now we give a necessary and sufficient condition for a diophantine equation with parameters to be solvable in every cyclotomic integer ring.

PROPOSITION 4.1. *Let $f(x, y)$ be a polynomial over $\mathbf{Z}$. The sentence $\forall x \exists y f(x, y)=0$ is true in $\mathbf{Z}[\zeta_m]$ for every odd prime m if and only if $f(x, y)$ has a linear factor of the form $y-g(x)$, where $g(x)$ is a polynomial over $\mathbf{Z}$.*

Proof. Clearly, if $g(x)$ is a polynomial over $\mathbf{Z}$, then for every element a of $\mathbf{Z}[\zeta_m]$, $g(a)$ is an element of $\mathbf{Z}[\zeta_m]$. Hence if $f(x, y)$ has a linear factor of the form $y-g(x)$, where $g(x)$ is a polynomial over $\mathbf{Z}$, then for every x' of $\mathbf{Z}[\zeta_m]$ there is a $y'=g(x')$ such that $f(x', y')=0$.

Conversely, we assume that $f(x, y)$ does not have factors of the form $y-g(x)$, where $g(x)$ is a polynomial over $\mathbf{Z}$. Let $f(x, y)=f_0(x, y)\prod_{i=1}^{r}(y-g_i(x))\times\prod_{j=1}^{t}(y-h_j(x))$, where $g_i(x)\in\mathbf{Q}[x]-\mathbf{Z}[x]$, $h_j(x)\in\mathbf{C}[x]-\mathbf{Q}[x]$ and $f_0(x, y)$ has no linear factors in $\mathbf{C}[x]$. We want to show that there is an m such that $\forall x \exists y f(x, y)=0$ is not true in $\mathbf{Z}[\zeta_m]$. Every polynomial $h_j(x)$ has a coefficient which is not a rational number. First, we require that the number m satisfy the condition that $h_j(x)$ does not belong to $\mathbf{Q}(\zeta_m)[x]$ for every j. Now let a_i be the least positive integer such that $a_i \cdot g_i(x)$ belongs to $\mathbf{Z}[x]$ for every $1 \leqq i \leqq r$. Let $a_i=\prod_{k=1}^{S} P_{i,k}^{t_{i,k}}$, where $P_{i,k}$ are different prime numbers in $\mathbf{N}$ and $t_{i,k}$ are the power of $P_{i,k}$ in a_i. For every integer n of $\mathbf{Z}[\zeta_m]$, let $\mathbf{N}_i(n)$ be the number of solutions of the congruence equation $a_i \cdot g_i(x)\equiv 0 \pmod{n}$ in $\mathbf{Z}[\zeta_m]$. It is well known [19, Corollary to Thm. 7.2] that $P_{i,k}$ $(1 \leqq k \leqq s)$ are also relatively prime in $\mathbf{Z}[\zeta_m]$ for every i. The Chinese Remainder Theorem is valid in every algebraic integer ring. With the same arguments as the case over $\mathbf{Z}$ [cf. 8], we obtain that $\mathbf{N}_i(a_i)=\prod_k \mathbf{N}_i(P_{i,k}^{t_{i,k}})$ for every i. Let $R(n)$ be the number of the elements of the residue class $\mathbf{Z}[\zeta_m]/(n)$ for every n of $\mathbf{Z}$. We claim that for any $\varepsilon>0$, we can find a suitable m such that $\mathbf{N}_i(a_i)/R(a_i)<\varepsilon$ for every i in $\mathbf{Z}[\zeta_m]$. In order to simplify the notations, for the remainder of this proof we fix an i and drop the i from the index. Thus we want to show that for a suitable m, $\mathbf{N}(a)/R(a)<\varepsilon$ in $\mathbf{Z}[\zeta_m]$. We first assume that a is a prime number in $\mathbf{Z}$. Let m be a large prime number in $\mathbf{Z}$ such that the *order h of a in the residue system modulo m*, i.e., h is the least positive integer such that $a^h \equiv 1 \pmod{m}$, is sufficiently large. It is well known [19, Thm. 8.7] that the ideal (a) decomposes in $\mathbf{Z}[\zeta_m]$ into $g=\varphi(m)/h$ different prime ideals of degree h and multiplicity $e=1$, where $\varphi(m)$ is the *Euler's φ-function*. Let A_l, $1 \leqq l \leqq g$, be the different prime factors of the ideal (a) in $\mathbf{Z}[\zeta_m]$. By the Chinese Remainder Theorem again, $\mathbf{N}(a)=\prod_{l=1}^{g}\mathbf{N}(A_l)$ and $R(a)=\prod_{l=1}^{g}R(A_l)=a^{hg}$. Since $a \cdot g(x)$ is a primitive polynomial over $\mathbf{Z}$, $a \cdot g(x)$ is also a primitive polynomial over $\mathbf{Z}[\zeta_m]$. For every l, the polynomial $a \cdot g(x)$ modulo the ideal A_l is not identically equal to zero. The number of the solutions of the congruence equation $a \cdot g(x)\equiv 0 \pmod{A_l}$ is less than or equal to d, the degree of $g(x)$, because $\mathbf{Z}[\zeta_m]/A_l$ is a field for every l. Then $\mathbf{N}(a)$ is at most d^g. Therefore $\mathbf{N}(a)/R(a)\leqq d^g/a^{h\cdot g}=(d/a^h)^g<\varepsilon$ if h is sufficiently large. Now let $a=\prod_{k=1}^{S}P_k^{t_k}$. Then

$$\begin{aligned}\mathbf{N}(a)/R(a) &= \prod_k \mathbf{N}(P_k^{t_k}) \Big/ \prod_k R(P_k^{t_k}) \\ &= \prod_k (\mathbf{N}(P_k^{t_k})/R(P_k^{t_k})) \\ &\leqq \prod_k (\mathbf{N}(P_k)/R(P_k))<\varepsilon^k.\end{aligned}$$

Let $\varepsilon<1/r$. Then $\sum_{i=1}^{r}\mathbf{N}_i(a_i)/R(a_i)=D<1$; D is the density of elements t of $\mathbf{Z}[\zeta_m]$ such that there exists an $i(1 \leqq i \leqq r)$; and $g_i(t)$ is an element of $\mathbf{Z}[\zeta_m]$. This implies that there is an a of $\mathbf{Z}[\zeta_m]$ such that $g_i(a)$ is not an element of $\mathbf{Z}[\zeta_m]$ for every i, $1 \leqq i \leqq r$. By Theorem A it follows that $\forall x \exists y f(x, y)=0$ is not true in $\mathbf{Z}[\zeta_m]$. This completes our proof. □

Now we extend this result to equations with parameters of more than 1.

COROLLARY 4.2. *Let* $f(x_1, \cdots, x_n, y)$ *be a polynomial over* $\mathbf{Z}$. *The sentence* $\forall x_1, \cdots, \forall x_n \exists y f(x_1, \cdots, x_n, y) = 0$ *is true in* $\mathbf{Z}[\zeta_m]$ *for every odd prime m if and only if* $f(x_1, \cdots, x_n, y)$ *has a linear factor of the form* $y - g(x_1, \cdots, x_n)$, *where* $g(x_1, \cdots, x_n)$ *is a polynomial over* $\mathbf{Z}$.

Proof. Clearly, if $f(x_1, \cdots, x_n, y)$ has a factor of the form $y - g(x_1, \cdots, x_n)$, then $\forall x_1 \cdots \forall x_n \exists y f(x_1, \cdots, x_n, y) = 0$ is true in $\mathbf{Z}[\zeta_m]$ for every m.

Conversely, we assume that $f(x_1, \cdots, x_n, y)$ does not have factors of the form $y - g(x_1, \cdots, x_n)$, where $g(x_1, \cdots, x_n)$ is a polynomial over $\mathbf{Z}$. Let $f(x_1, \cdots, x_n, y) = f_0(x_1, \cdots, x_n, y) \prod_{i=1}^{r} (y - g_1(x_i, \cdots, x_n)) \prod_{j=1}^{t} (y - h_j(x_1, \cdots, x_n))$, where $g_i(x_1, \cdots, x_n) \in \mathbf{Q}[x_1, \cdots, x_n] - \mathbf{Z}[x_1, \cdots, x_n]$, $h_j(x_1, \cdots, x_n) \in \mathbf{C}[x_1, \cdots, x_n] - \mathbf{Q}[x_1, \cdots, x_n]$, and $f_0(x_1, \cdots, x_n, y)$ has no linear factors in $\mathbf{C}[x_1, \cdots, x_n]$. First, let m be a prime number such that $h_j(x_1, \cdots, x_n)$ does not belong to $\mathbf{Q}(\zeta_m)[x_1, \cdots, x_n]$ for every $j \leqq t$. Let d be a positive number such that d is greater than the degree of $g_i(x_1, \cdots, x_n)$ with respect to x_k for every $k \leqq n$ and $i \leqq r$. Let $G_i(x)$ be $g_i(x, x^d, x^{d^2}, \cdots, x^{d^{n-1}})$. (This is called Kronecker's substitution.) We can see that the coefficients of $G_i(x)$ are the same as the coefficients of $g_i(x_1, \cdots, x_n)$. Therefore, $G_i(x) \in \mathbf{Q}[x] - \mathbf{Z}[x]$. With the same arguments as in the proof of Proposition 4.1, we can choose a number m so that there exists an element b of $\mathbf{Z}[\zeta_m]$; $G_i(b)$ is not an element of $\mathbf{Z}[\zeta_m]$ for every $1 \leqq i \leqq r$. This implies that there exist integers $b_1 = b$, $b_2 = b^d, \cdots, b_n = b^{d^{n-1}}$ such that $g_i(b_1, b_2, \cdots, b_n)$ is not an element of $\mathbf{Z}[\zeta_m]$ for every i, $1 \leqq i \leqq r$. By Theorem A, $\forall x_1 \cdots \forall x_n \exists y f(x_1, \cdots, x_n, y) = 0$ is not true in $\mathbf{Z}[\zeta_m]$. This completes our proof. □

THEOREM 4.3. $\forall^n \exists$ EQUATION OVER ALL AIR *is in P.*

Proof. Let $f(x_1, \cdots, x_n, y)$ be a polynomial over $\mathbf{Z}$. All we need to do is to factor $f(x_1, \cdots, x_n, y)$ over $\mathbf{Z}$. It is well known [14] that this can be done in polynomial time. Then by Proposition 4.2, $\forall x_1 \cdots \forall x_n \exists y f(x_1, \cdots, x_n, y) = 0$ is true in every algebraic integer ring if and only if $f(x_1, \cdots, x_n, y)$ has a factor of the form $y - g(x_1, \cdots, x_n)$, where $g(x_1, \cdots, x_n)$ is a polynomial over $\mathbf{Z}$. This completes our proof. □

Now with Proposition 4.1 we prove a result which is similar to Proposition 3.4. Notice that we do not need to prove the corresponding part (b); it is automatically true.

PROPOSITION 4.4. *Let* $\varphi(x, y)$ *be a formula in the form* $\bigvee_{i=1}^{S} [\bigwedge_{j=1}^{m_i} f_{i,j}(x, y) = 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(x, y) \neq 0]$, *where* $f_{i,j}(x, y)$ *and* $g_{i,k}(x, y)$ *are polynomials over* $\mathbf{Z}$. *If the sentence* $\forall x \exists y\, \varphi(x, y)$ *is true in every algebraic integer ring (or* $\mathbf{Z}[\zeta_m]$ *for every odd prime m) then there exist an i* ($1 \leqq i \leqq s$) *and a polynomial* $F(x)$ *over* $\mathbf{Z}$ *such that* $y - F(x)$ *is a common factor of each* $f_{i,j}(x, y)$ ($1 \leqq j \leqq m_i$), *but not a factor of any* $g_{i,k}(x, y)$ ($1 \leqq k \leqq n_i$) *over the ring* $\mathbf{Z}[x, y]$.

Proof. First, if for some i, j, and k the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ have common factors over $\mathbf{Z}[x, y]$, then we can omit the common factors from the polynomial $f_{i,j}(x, y)$. Hence we may assume that for every i, j, and k the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime in $\mathbf{Z}[x, y]$. For every x and y of an algebraic integer ring I, if $\varphi(x, y)$ is true, then $\bigvee_{i=1}^{S} f_{i,1}(x, y) = 0$ or equivalently, $[\prod_{i=1}^{S} f_{i,1}(x, y)] = 0$ is true in I. The sentence $\forall x \exists y\, \varphi(x, y)$ is true in $\mathbf{Z}[\zeta_m]$ for every odd prime m, therefore $\forall x \exists y\, [\prod_{i=1}^{S} f_{i,1}(x, y)] = 0$ is true in $\mathbf{Z}[\zeta_m]$ for every odd prime m. By Proposition 4.1, there exists a polynomial $F_1(x)$ over $\mathbf{Z}$ such that $y - F_1(x)$ is a factor of the polynomial $\prod_i f_{i,1}(x, y)$. Since $y - F_1(x)$ is an irreducible polynomial in $\mathbf{Z}[x, y]$, $y - F_1(x)$ must be a factor of the polynomial $f_{l,1}(x, y)$ for some ($1 \leqq l \leqq s$) in $\mathbf{Z}[x, y]$. Because the polynomials $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime, $y - F_1(x)$ is not a factor of any one of the polynomials $g_{l,k}(x, y)$ for $1 \leqq k \leqq n_l$. Now suppose that $y - F_1(x)$ is not a factor of one of the polynomials $f_{l,j}(x, y)$ ($1 \leqq j \leqq m_l$). Let A be the set of x's of the common

roots of the equations $y-F_1(x)=0$ and $f_{l,j}(x,y)=0$ in the complex number field $\mathbf{C}$. A is a finite set. In order to make $\forall x \exists y\, \varphi(x,y)$ true in every algebraic integer ring I, for every x of $I-A$ there exists a y of I such that $[\prod_i f_{i,1}(x,y)/(y-F_1(x))]=0$. Since A is finite, the density of $I-A$ in I equals 1, and $I-A$ intersects every arithmetic progression in I. From the proof of Proposition 4.1, we obtain that there exists a factor $y-F_2(x)$ of the polynomial $\prod_i f_{i,1}(x,y)/(y-F_1(x))$. The polynomial $F_2(x)$ might not satisfy the conditions of the conclusion. Then with the same arguments there must exist another factor $y-F_3(x)$ of the polynomial $\prod_i f_{i,1}(x,y)/(y-F_1(x))\cdot(y-F_2(x))$. Since the degree of y in the polynomial $\prod_i f_{i,1}(x,y)$ is finite, there must exist a polynomial $F(x)$ satisfying the conditions of the conclusion. □

Now we prove another main theorem.

THEOREM 4.5. ∀∃ SENTENCE OVER ALL AIR *is in P.*

Proof. Let $\varphi(x,y)$ be a DNF formula containing no other free variables. We write that $\varphi(x,y)=\bigvee_{i=1}^{S}[\bigwedge_{j=1}^{m_i} f_{i,j}(x,y)=0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(x,y)\neq 0]$ where $f_{i,j}(x,y)$ and $g_{i,k}(x,y)$ are polynomials over $\mathbf{Z}$.

(S_1) We factor every $f_{i,j}(x,y)$ and $g_{i,k}(x,y)$ in $\mathbf{Z}[x,y]$. This can be done in polynomial time [14]. By Proposition 4.4, if $\forall x \exists y\, \varphi(x,y)$ is true in every algebraic integer ring, then there exist an i ($1 \leqq i \leqq s$) and a polynomial $F(x)$ over $\mathbf{Z}$ such that $y-F(x)$ is a common factor of each $f_{i,j}(x,y)$ ($1\leqq j \leqq m_i$) but not a factor of any $g_{i,k}(x,y)$ ($1 \leqq k \leqq n_i$) over the ring $\mathbf{Z}[x,y]$. Let S be the set of these polynomials $F(x)$. The number of the elements of S is less than $\sum_i d_i$, where $d_i=\min_j d_{i,j}$, $d_{i,j}$ is the *degree* of y in the polynomial $f_{i,j}(x,y)$. Assume that S is nonempty; otherwise $\forall x \exists y\, \varphi(x,y)$ is false in some algebraic integer rings. Notice that for every element a of an algebraic integer ring I, $F(a)$ is an element of I for every $F(x)$ of S. Since $F(x)$ is a common factor of the polynomials $f_{i,j}(x,y)$ ($1 \leqq j \leqq m_i$), $\forall x \exists y \bigwedge_{j=1} f_{i,j}(x,y)=0$ is true in every algebraic integer ring. The only possible case that $\forall x \exists y\, \varphi(x,y)$ might be false in some algebraic integer ring I is that there exist a k ($1 \leqq k \leqq n_i$) and an a of I such that $g_{i,k}(a,F(a))=0$.

(S_2) We factor all the polynomials $g_{i,k}(x,F(x))$, where $F(x)$ is the common factor of $f_{i,j}(x,y)$ ($1 \leqq j \leqq m_i$), over $\mathbf{Z}$ for $1 \leqq i \leqq s$, $1 \leqq k \leqq n_i$. This can be done in polynomial time [16]. Let T be the set of all monic irreducible factors of the polynomials $g_{i,k}(x,F(x))$. Let $x-a_l$, $1 \leqq l \leqq r$, be the polynomial in T with degree 1. We apply the algorithm ∃ SENTENCE to check the sentence $\exists y\, \varphi(a_l,y)$ in $\mathbf{Z}$ for every a_l ($1 \leqq l \leqq r$). If there is an a_l such that $\exists y\, \varphi(a_l,y)$ is false in $\mathbf{Z}$, then $\forall x \exists y\, \varphi(x,y)$ is false in $\mathbf{Z}$. Then sentence $\forall x \exists y\, \varphi(x,y)$ does not hold in every algebraic integer ring. If for every a_l the sentence $\exists y\, \varphi(a_l,y)$ is true in $\mathbf{Z}$, then $\forall x \exists y\, \varphi(x,y)$ is true in $\mathbf{Z}$. Now we assume that this is the case.

(S_3) We apply the algorithm ∃ SENTENCE to check the sentence $\exists y\, \varphi(t,y)$ in the ring of integers of the field $\mathbf{Q}[t]/g(t)$ for every polynomial $g(t)$ with degree greater than 1 of T. Here we change the variable x of $g(x)$ in T to variable t and take t to be an element of $\mathbf{Q}[t]/g(t)$. If $\exists y\, \varphi(t,y)$ is false in some rings of integers then, of course, $\forall x \exists y\, \varphi(x,y)$ is false in some algebraic integer rings. We claim that if $\exists y\, \varphi(t,y)$ is true in the ring of integers of $\mathbf{Q}[t]/g(t)$ for every $g(t)$, then $\forall x \exists y\, \varphi(x,y)$ is true in every algebraic integer ring. Suppose that this is not the case. Let I be an algebraic integer ring and $\forall x \exists y\, \varphi(x,y)$ is false in I. Then there exists a b of I such that $\exists y\, \varphi(b,y)$ is false in I. This implies that $g_{i,k}(b,F(b))=0$ for certain i, k and $F(x)$ of S. Then b must be a root of the equation $g(x)=0$ for a $g(x)$ of T. Clearly, $\mathbf{Q}(b)\simeq \mathbf{Q}[t]/g(t)$. Let A be the ring of integers of the field $\mathbf{Q}(b)$. Every element of A is an algebraic integer and can be represented by the form $\sum a_i b^i$; a_i are elements of $\mathbf{Q}$. The algebraic integer ring I contains b; A is a subring of I. The sentence $\exists y\, \varphi(b,y)$ is true in A, since $\exists y\, \varphi(t,y)$ is true in the ring of integers of $\mathbf{Q}[t]/g(t)$ and $\mathbf{Q}(b)\simeq Q[t]/g(t)$.

The sentence $\exists y\, \varphi(b, y)$ is true in I, since A is a subring of I. This is a contradiction and proves our claim. All the steps (S_1), (S_2), and (S_3) can be done in polynomial time. Therefore ∀∃ SENTENCE OVER ALL AIR is in P if the given formula is in DNF.

Now we prove the case $\varphi(x, y)$ in CNF, i.e., $\varphi(x, y) = \bigwedge_{i=1}^{S} [\bigvee_{j=1}^{m_i} f_{i,j}(x, y) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y) \neq 0]$, where $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are polynomials over $\mathbf{Z}$.

(T_1) Just as we did in the step (T_1) of Theorem 3.6, we omit from $g_{i,k}(x, y)$ the GCD of $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ for every i, j, and k. Hence we may assume that for every i, $f_{i,j}(x, y)$ and $g_{i,k}(x, y)$ are relatively prime for $1 \leqq j \leqq m_i$ and $1 \leqq k \leqq n_i$ in the above formula.

(T_2) First, we assume that for every i there is a k such that $g_{i,k}(x, y) \not\equiv 0$. Given a polynomial $g_{i,k}(x, y) = h_m(x)y^m + \cdots + h_0(x)$, let $G_{i,k}(x)$ be the GCD of polynomials $\{h_m(x), \cdots, h_0(x)\}$ and $G_i(x)$ be the GCD of $G_{i,k}(x)$ for $k \leqq n_i$. Now we factor $G_i(x)$ over $\mathbf{Z}$ by the algorithm of [16] and let S be the set of all monic irreducible factors of the polynomials $G_i(x)$ for $1 \leqq i \leqq s$. Let $x - a_\alpha$, $1 \leqq \alpha \leqq r$, be the polynomials in S with degree 1. We apply the algorithm ∃ SENTENCE to check $\exists y\, \varphi(a_\alpha, y)$ over $\mathbf{Z}$ for every $\alpha \leqq r$ and $\exists y\, \varphi(t, y)$ over the ring of integers of the field $\mathbf{Q}[t]/g(t)$ for every polynomial $g(x)$ of S with degree greater than I. If $\exists y\, \varphi(a_\alpha, y)$ is false in $\mathbf{Z}$ for an a_α or $\exists y\, \varphi(t, y)$ is false in the algebraic integer ring of $\mathbf{Q}[t]/g(t)$ for a $g(x)$ then, of course, $\forall x \exists y\, \varphi(x, y)$ cannot be true in every algebraic integer ring. Otherwise, we claim that $\forall x \exists y\, \varphi(x, y)$ is true in every algebraic integer ring. Suppose that this is not the case. Let I be an algebraic integer ring and $\forall x \exists y\, \varphi(x, y)$ is false in I. Let b be an element of I such that $\forall y \bigvee_{i=1}^{S} [\bigwedge_{j=1}^{m_i} f_{i,j}(b, y) \neq 0 \wedge \bigwedge_{k=1}^{n_i} g_{i,k}(b, y) = 0]$ is true. We obtain that there is an i, $G_i(b) = 0$. Since b is an algebraic integer, there is a $g(x)$ in S such that $g(b) = 0$. Let A be the algebraic integer ring of $\mathbf{Q}(b)$. Since $\mathbf{Q}(b) \simeq \mathbf{Q}[x]/g(x)$ and A is a subring of I, $\exists y \varphi(b, y)$ is true in I. But this contradicts our assumption. Therefore, $\forall x \exists y\, \varphi(x, y)$ is true in every algebraic integer ring. Now we assume that for some i, $g_{i,k}(x, y) \equiv 0$ for every $k \leqq n_i$. Since $a = 0 \vee b = 0 \Leftrightarrow a \cdot b = 0$ is true in every algebraic integer ring, we may combine several equations into one and write that

$$\varphi(x, y) = \bigwedge_{i=1}^{s} \left[\bigvee_{j=1}^{m_i} f_{i,j}(x, y) = 0 \vee \bigvee_{k=1}^{n_i} g_{i,k}(x, y) \neq 0 \right] \wedge \bigwedge_{l=1}^{p} F_l(x, y) = 0.$$

(T_3) We apply the Euclidean algorithm to find the GCD, $G(x, y)$, of polynomials $F_l(x, y)$ for $1 \leqq l \leqq p$, then factor $G(x, y)$ over $\mathbf{Z}$ by Lenstra's algorithm [14]. We may write $G(x, y) = G_0(x, y) \prod_{\beta=1}^{q} (y - G_\beta(x))$, where $G_0(x, y)$ has no factor of the form $y - G(x)$. If $q = 0$, i.e., $G(x, y)$ has no factor of the form $y - G(x)$, then $\forall x \exists y \bigwedge_{l=1}^{p} F_l(x, y) = 0$ is false in some algebraic integer ring by Proposition 4.4. Therefore, $\forall x \exists y\, \varphi(x, y)$ is false in some algebraic integer ring. Now we assume that $q \neq 0$.

(T_4) Let $H_{i,k,\beta}(x) = g_{i,k}(x, G_\beta(x))$ and T_i be the set of polynomials $G_\beta(x)$ such that $H_{i,k,\beta}(x) \equiv 0$ for every $k \leqq n_i$ and $T = \bigcup_i T_i$. Let $G'(x, y) = G_0(x, y) \prod_{\delta=1}^{r} (y - G_\delta(x))$, where $G'(x, y)$ is the polynomial omitting from $G(x, y)$ any factor $y - G(x)$ if $G(x) \in T$. With similar arguments as we made in the step (T_4) of Theorem 3.6 we obtain that $r \neq 0$, i.e., $G'(x, y)$ has a factor of the form $y - G(x)$, where $G(x)$ is a polynomial with coefficients in $\mathbf{Z}$. Notice that for any algebraic integer ring I, $G(a)$ is in I if a is in I. This time we do not need the nondeterministic algorithm in the step (T_4) of Theorem 3.6.

(T_5) Let $G(x) \in \mathbf{Z}[x]$ and $y - G(x)$ be a factor of $G'(x, y)$. We factor the polynomial $g_{i,k}(x, G(x))$ over $\mathbf{Z}$ for every $i \leqq s$ and $k \leqq n_i$. Let T be the set of all monic irreducible factors of the polynomials $g_{i,k}(x, G(x))$. As in step (T_2), we check the

sentence $\exists y\, \varphi(a, y)$ over $\mathbf{Z}$ if $x-a$ is in T and $\exists y\, \varphi(x, y)$ over the algebraic integer ring of $\mathbf{Q}[x]/g(x)$ if $g(x)$ is in T and the degree of $g(x)$ is greater than 1. If there is a sentence $\exists y\, \varphi(a, y)$ or $\exists y\, \varphi(x, y)$ that is false, then $\forall x \exists y\, \varphi(x, y)$ cannot be true in every algebraic integer ring. Otherwise, by the same arguments, $\forall x \exists y\, \varphi(x, y)$ is true in every algebraic integer ring. This completes our proof for the case $\varphi(x, y)$ in CNF. Therefore $\forall\exists$ SENTENCE OVER ALL AIR is in P. □

With similar arguments as those used to prove Theorem 4.5, we can prove that the decision problems of $\forall\exists$ sentences true in some other classes of rings of algebraic integers are also in P. We demonstrate the case of radical integer rings (RIR).

COROLLARY 4.6. *$\forall\exists$ SENTENCE OVER ALL RIR is in P.*

Proof. Every cyclotomic extension is a radical extension, therefore we can apply Proposition 4.4. Our algorithms are similar to the algorithms that decide $\forall\exists$ SENTENCE OVER ALL AIR. We sketch the proof for the formula in DNF only. The only difference is in (S_2). Here let T' be the subset of T such that all the polynomials of T' are also solvable by radicals. It is well known [12] that solvability by radicals is in polynomial time. We can obtain this set T' in polynomial time. Notice that the field $\mathbf{Q}[t]/g(t)$ (not the splitting field of $g(x)$) is a radical extension field. If $\exists y\, \varphi(t, y)$ is false in the ring of integers of $\mathbf{Q}[t]/g(t)$, then $\forall x \exists y\, \varphi(x, y)$ is false in some radical integer rings. Now we need to prove that if $\exists y\, \varphi(t, y)$ is true in the ring of integers of $\mathbf{Q}[t]/g(t)$ for every $g(x)$ of T', then $\forall x \exists y\, \varphi(x, y)$ is true in every radical integer ring. Suppose that this is not the case. Let R be a radical integer ring and $\forall x \exists y\, \varphi(x, y)$ is false in R. Then there exists a b of R such that $\exists y\, \varphi(b, y)$ is false in R. This implies that there exist i, k and $F(x)$ of S such that $g_{i,k}(b, F(b))=0$. Then b must be a root of the equation $g(x)=0$ for a $g(x)$ of T. Since R is a radical integer ring and b is an element of R, b can be expressed in terms of radicals. The irreducible polynomial $g(x)$ has a root expressed in terms of radicals, so $g(x)$ is solvable by radicals [13]. Hence $g(x)$ must be an element of T'. Clearly, $\mathbf{Q}(b) \simeq \mathbf{Q}[t]/g(t)$, let A be the ring of integers of $\mathbf{Q}(b)$. Then A is a subring of R. The sentence $\exists y\, \varphi(b, y)$ is true in A. It is also then true in R. This contradicts that $\exists y\, \varphi(b, y)$ is false in R, and completes our proof. □

Let $f(x)$ be a polynomial over $\mathbf{Z}$. Then $f(x)$ can be checked for normality in polynomial time. Furthermore, if $f(x)$ is normal, then computing its Galois group can be done in polynomial time [11]. Therefore given a polynomial $f(x)$, we can check in polynomial time whether the splitting field of $f(x)$ over $\mathbf{Q}$ is a *cyclic* (or *abelian*) extension field or not. Because the splitting field of $f(x)$ is an abelian extension only if $f(x)$ is normal. With similar arguments as those used to prove Corollary 4.6, we obtain the following corollaries.

COROLLARY 4.7. *$\forall\exists$ SENTENCE OVER ALL CIR is in P.*

COROLLARY 4.8. *$\forall\exists$ SENTENCE OVER ALL BIR is in P.*

Acknowledgment. The author would like to express his sincere thanks to two unknown referees. They pointed out a mistake in a previous version of this paper and made many helpful suggestions.

REFERENCES

[1] A. BAKER, *Transcendental Number Theory*, Cambridge University Press, Cambridge, U.K., 1975.

[2] S. A. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[3] M. DAVIS, *Hilbert's tenth problem is unsolvable*, Amer. Math. Monthly, 80 (1973), pp. 233–269.

[4] J. DENEF, *Hilbert's tenth problem for quadratic rings*, Proc. Amer. Math. Soc., 48 (1975), pp. 214–220.

[5] ———, *Diophantine sets over algebraic integer rings* II, Trans. Amer. Math. Soc., 257 (1980), pp. 227–236.

[6] J. DENEF AND L. LIPSHITZ, *Diophantine sets over some rings of algebraic integers*, J. London Math. Soc. (2), 18 (1978), pp. 385-391.

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[8] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fifth Edition, Oxford University Press, London, U.K., 1979.

[9] E. KALTOFEN, *Polynomial-time reductions from multivariate to bi- and univariate integral polynomial factorization*, SIAM J. Comput., 12 (1985), pp. 469-489.

[10] R. KANNAN, A. K. LENSTRA, AND L. LOVÁSZ, *Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers*, Math. Comp., 50 (1988), pp. 235-250.

[11] S. LANDAU, *Factoring polynomials over algebraic number fields*, SIAM J. Comput., 14 (1985), pp. 184-195.

[12] S. LANDAU AND G. L. MILLER, *Solvability by radical is in polynomial time*, J. Comput. System Sci., 30 (1985), pp. 179-208.

[13] S. LANG, *Algebra*, Addison-Wesley, Reading, MA, 1971.

[14] A. K. LENSTRA, *Factoring multivariate integral polynomials*, Theoret. Comput. Sci., 34 (1984), pp. 207-213.

[15] ———, *Factoring multivariate polynomials over algebraic number fields*, SIAM J. Comput., 16 (1987), pp. 591-598.

[16] A. K. LENSTRA, H. W. LENSTRA, AND L. LOVÁSZ, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 515-534.

[17] H. W. LENSTRA, Lectures at Bonn Workshop on Foundations of Computing, Bonn, Federal Republic of Germany, 1987.

[18] K. MANDERS AND L. ADLEMAN, NP-*complete decision problem for binary quadratics*, J. Comput. System Sci., 16 (1978), pp. 168-184.

[19] H. MANN, *Introduction to Algebraic Number Theory*, Ohio State University Press, Columbus, OH, 1955.

[20] M. MIGNOTTE, *An inequality about factors of polynomials*, Math. Comp., 28 (1974), pp. 1153-1157.

[21] J. ROBINSON, *Definability and decision problems in arithmetic*, J. Symbolic Logic, 14 (1949), pp. 98-114.

[22] ———, *The undecidability of algebraic rings and fields*, Proc. Amer. Math. Soc., 10 (1959), pp. 950-957.

[23] ———, *On the decision problem for algebraic rings*, Studies in Mathematical Analysis and Related Topics, No. 42, Stanford University Press, Stanford, CA, 1962, pp. 297-304.

[24] A. SCHINZEL, *Diophantine equations with parameters*, in Journées Arithmetiques 1980, J. V. Armitage, ed., London Math. Soc. Lecture Note Ser. 56, Cambridge University Press, Cambridge, U.K., 1982, pp. 211-217.

[25] ———, *Selected Topics on Polynomials*, The University of Michigan Press, Ann Arbor, MI, 1982.

[26] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time*, in Proc. 5th Annual ACM Symposium on Theory of Computing, 1973, pp. 1-9.

[27] S. P. TUNG, *On weak number theories*, Japan. J. Math., 11 (1985), pp. 203-232.

[28] ———, *Provability and decidability of arithmetical universal-existential sentences*, Bull. London Math. Soc., 18 (1986), pp. 241-247.

[29] ———, *Computational complexities of diophantine equations with parameters*, J. Algorithms, 8 (1987), pp. 324-336.

[30] ———, *Computational complexity of arithmetical sentences*, preprint.

[31] ———, *Decidable fragments of field theories*, J. Symbolic Logic, to appear.

[32] ———, *Polynomial time algorithms for sentences over number fields*, Inform. and Comput., to appear.

[33] P. WEINBERGER AND L. ROTHSCHILD, *Factoring polynomials over algebraic number fields*, ACM Trans. Math. Software, 2 (1976), pp. 335-350.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 144-148, February 1991
© 1991 Society for Industrial and Applied Mathematics
008

AN OPTIMAL ON-LINE ALGORITHM FOR K-SERVERS ON TREES*

MAREK CHROBAK† AND LAWRENCE L. LARMORE†

Abstract. The k-server problem is investigated when the metric space is a tree. For this case an on-line k-competitive algorithm for k-servers is presented. The competitiveness ratio k is optimal. The algorithm is memoryless, in the sense that it does not use any information from the past.

Key words. on-line algorithms, the server problem, competitive analysis, metric spaces, trees

AMS(MOS) subject classification. 68Q20

1. Introduction. Let M be a metric space. That is, for any two points $x, y \in M$ we are given their distance $\|x, y\| \geq 0$ such that $\|x, y\| > 0$ for $x \neq y$, and the triangle inequality is also satisfied: $\|x, y\| + \|y, z\| \geq \|x, z\|$ for all $x, y, z \in M$.

We are also given k-servers that can move among points of M. At each time slot, a request $x \in M$ appears, and we have to "serve" this request, that is, choose one of our servers and move it to x. Other servers are also allowed to move. Our measure of cost is the distance by which we move our servers. The problem is to design a strategy that minimizes the cost of servicing a sequence of requests given on-line. The server problem is an abstraction of several practical problems, including heuristics for linear search, paging, and motion planning for 2-headed disks. For example, in the paging (or, equivalently, caching) problem we have a two-level memory system with a total of n pages, k of which can reside in fast memory. Given a reference to a memory page x which is not currently in the fast memory, we remove some page y from the fast memory and replace it with x. Our objective is to minimize the number of page replacements. This can be modeled by a server problem on an n-point space where all distances are equal to 1. See [2], [7], and [9] for more references.

Suppose that we are given a sequence R of requests. A *schedule* is a specification of which server serves which request of R. Any schedule that achieves a minimum cost is called an *optimal* schedule. It is known (see [2]) how to compute an optimal schedule for R in polynomial time, assuming that the whole sequence R is given off-line, in advance.

However, no *on-line* algorithm can achieve the optimal cost on each sequence of requests. The current research on this problem concentrates on finding *c-competitive* on-line algorithms, that is, algorithms which compute a schedule of cost at most $c \cdot opt + b$, where opt is the cost of the optimal schedule and b is a constant that is allowed to depend only on the initial configuration. Manasse, McGeoch, and Sleator [7] proved that there is no on-line c-competitive algorithm for $c < k$, for any metric space with at least $k + 1$ points. The problem of whether there is a general on-line algorithm with competitiveness ratio k remains open.

For $k = 2$, the problem was solved by Manasse, McGeoch, and Sleator [7], who gave a 2-competitive algorithm for 2 servers. Irani and Rubinfeld [5] proved that a version of a balancing algorithm (dividing the work more or less equally among the servers) is 10-competitive for 2 servers. Raghavan and Snir [9] presented a randomized 2-competitive algorithm whose competitiveness ratio is between 3 and 6.

* Received by the editors July 24, 1989; accepted for publication (in revised form) April 11, 1990.
† Department of Computer Science, University of California, Riverside, California 92521.

Some research has been done for some specific metric spaces. Chrobak et al. [2] presented k-competitive algorithms for the line and the weighted cache problem, where the distance to x from any other point is $w(x)$, the weight of x. The weighted cache problem was also considered in [9], where a randomized k-competitive algorithm for this problem was given. Using randomization for the paging problem, it is possible to beat the lower bound k for deterministic algorithms. Fiat et al. [4] presented a randomized algorithm for this problem with ratio $2H_k$, improved later to H_k by McGeoch and Sleator in [8] (where H_k is the kth harmonic number). Very recently, Coppersmith et al. [3] have discovered a randomized algorithm with competitiveness constant k that works for a class of metric spaces called *resistive*. This class includes many known metric spaces, for example, trees.

Raghavan and Snir [9] introduced a notion of a *memoryless* algorithm, an algorithm whose behaviour does not depend on the past. A memoryless algorithm makes the decision based only on the current configuration of the servers and the position of the request.

In this paper, we present a k-competitive algorithm for k-servers on trees. By a *tree* we mean a metric space that is topologically equivalent to a free tree (in the graph-theoretic sense), and the distance is measured along the branches of the tree. The lower bound of k for k-servers presented in [7] also applies to the special case of a tree, and therefore this result is best possible for deterministic algorithms. The algorithm is also memoryless. This generalizes a similar result for the line in [2]. It should be pointed out here that, in the case of trees, some server algorithms with memory can be transformed into memoryless algorithms with the same asymptotic competitiveness ratio, by a careful encoding of the memory state in the binary expansion of the distances between the servers. This requires, however, complicated and expensive bit operations, unlike our algorithm, that is realized by a simple, piecewise linear function.

An interesting feature of this algorithm is that it can be applied to other problems, even though they may not have a tree structure at first glance. For example, consider the paging problem, where our metric space consists of n points where all distances are equal to 1. We "embed" this into a tree as follows: our tree is a star with n arms of length 0.5, and we place n points at the ends of those arms. By applying our algorithm to this star, we obtain a k-competitive algorithm for the paging problem. Quite surprisingly, this algorithm turns out to be equivalent to the Flush-When-Full cache strategy from [6]. In the same way, it also gives a k-competitive algorithm for a much more difficult problem: the *symmetric weighted cache*, where each point x is given some weight $w(x) \geq 0$, and the distance between x and y is $w(x) + w(y)$. It is easy to see that the symmetric and nonsymmetric version of weighted cache are almost equivalent: on any sequence of requests their optimal costs differ by at most half of the difference of weights of the initial and final configuration.

We also show that our result gives an algorithm for an arbitrary n-point metric space with competitiveness ratio $k(n-1)$. This improves the previously known bound of $2\binom{n}{k} - 1$ that can be derived from the work of Borodin, Linial, and Saks [1]. For some specific metric spaces this ratio can be made yet smaller.

2. The adversary and potential methods. The adversary method is used very often for proving lower bounds on the complexity of various computational problems. We employ some of the ideas of this method for establishing the *upper* bound on the competitiveness ratio. In the proofs, we view the computation as a game between our servers $s_1, \cdots, s_k$ and adversary's servers $a_1, \cdots, a_k$, and measure the ratio

between S, the work done by our servers, and A, the work of the adversary's servers. The goal is to show that independently of the way the adversary moves, we have $S \leq cA + b$. This implies immediately that the algorithm is c-competitive, since one of the adversary's computations will correspond to the optimal schedule. The same approach has been used in other proofs of competitiveness.

The potential method has been used before in the analysis of the amortized complexity of various data structures. The technique can be viewed as follows. Given an on-line server algorithm, we define a potential function Φ that maps any possible configuration of the servers (ours and the adversary's) to a nonnegative real number. Let Φ^t denote the value of the potential function after t steps of the algorithm.

The adversary starts from the same initial configuration as our servers. We assume that Φ^0, the initial value of the potential, depends only on this initial configuration. Additionally, we assume that at each step of the game, the adversary first moves a single server to the request point, and then we apply our algorithm to serve the request. It is not hard to prove that those restrictions do not lead to loss of generality (see [7]). Let $\widetilde{\Phi}^t$ denote the value of the potential function after $t-1$ steps and one additional adversary's move. A^t and S^t denote, respectively, the adversary's and our costs at step t. Our technique relies on the following lemma, whose simple proof is left to the reader.

LEMMA 2.1. *Suppose that for each step $t \geq 1$ we have the following:*
(i) $\widetilde{\Phi}^t - \Phi^{t-1} \leq \alpha A^t$, *and*
(ii) $\widetilde{\Phi}^t - \Phi^t \geq \beta S^t$.
Then $S \leq (\alpha/\beta)A + \Phi^0$, that is, our algorithm is c-competitive for $c = \alpha/\beta$.

3. The competitive algorithm for trees. In this paper, by a *tree* we understand a planar embedding of a free tree (in the graph-theoretic sense). If T is a tree, then the distance $\|x, y\|$ is the arc-length of the unique simple path through T from x to y. This path, with x excluded, is denoted by $(x, y]$, and called an *interval.*

For simplicity, s_p and a_i will also denote the current positions of the servers s_p, a_i. If the request is on point x, then we call our server s_p *active*, if there are no more of our servers in the interval $(s_p, x]$. If several servers occupy the same position as s_p, and all satisfy the condition above, then only *one* of them is chosen arbitrarily as the active one; the others are not. Our algorithm works as follows.

ALGORITHM 1. Move all of our active servers continuously with the same speed towards x until one of them (obviously the closest one) reaches the request. Note that during this motion some active servers may become nonactive, and then they halt.

More precisely, the algorithm can be formulated as follows.

while none of our servers is on x **do begin**
  let $d = \min_p \|s_p, y_p\|$, where $y_p \in (s_p, x]$ is either a vertex or x;
  move each active server s_p by d towards x
end.

Our potential function is defined now by

$$\Phi = k\|M_{\min}\| + \sum_{p<q} \|s_p s_q\|,$$

where $M_{\min}$ is a minimum weight matching in a bipartite graph with components $\{s_1, \cdots, s_k\}$ and $\{a_1, \cdots, a_k\}$, where the weight of edge (s_p, a_i) is $\|s_p a_i\|$.

LEMMA 3.1. $\widetilde{\Phi}^t - \Phi^{t-1} \leq kA^t$.

Proof. Since the adversary moves only one server, only one edge in $M_{\min}$ can change. If this server moves by d, then the potential increases by at most kd. This implies the lemma. □

LEMMA 3.2. $\widetilde{\Phi}^t - \Phi^t \geq S^t$.

Proof. Without loss of generality we assume that the request is on a_1. Assume that the servers $s_1, \cdots, s_q$ are active. It is easy to see that there is a minimum matching $M_{\min}$ in which a_1 is matched to one of $s_1, \cdots, s_q$. Suppose that all our servers moved by d. Then the weight of a minimum matching cannot increase by more than $(q-2)d$. For $p = 1, \cdots, q$, let ℓ_p be the number of our servers s_r such that s_p is between s_r and a_1. When s_p moves by d, its distance to $l_p - 1$ of our servers increases by d, but the distances to $k - \ell_p$ of our servers decreases by d. Therefore, during this movement the change of the potential is at most

$$\begin{aligned} k(q-2)d + \sum_{p=1}^{q}(\ell_p - 1 - k + \ell_p)d &= \left[k(q-2) + 2\sum_{p=1}^{q}\ell_p - q - kq\right]d \\ &= \left[k(q-2) + 2k - q - kq\right]d \\ &= -qd, \end{aligned}$$

and our cost is qd. This proves the lemma. □

From Lemmas 3.1 and 3.2, using Lemma 2.1 we immediately obtain the following result.

THEOREM 3.3. *Algorithm* 1 *is k-competitive.*

4. Applications. As was shown in the Introduction, our algorithm can be applied directly to other metric spaces that can be "embedded" in a tree. Another example of such spaces are so-called *ultrametric spaces.* A metric space M is called *ultrametric* if for any $x, y, z \in M$ we have $\|xy\| \leq \max\{\|xz\|, \|yz\|\}$. It is easy to show that each ultrametric space can be isometrically embedded into a tree, so our algorithm can be applied to ultrametric spaces as well.

In this section we show that this algorithm can also be used to obtain an algorithm that is $k(n-1)$-competitive on every n-point metric space.

Let M be a metric space with n points.

ALGORITHM 2. Fix a minimum spanning tree T of M, and apply Algorithm 1 pretending that T is the underlining metric space.

By using the subscript T we will distinguish the distances and costs in T from those in M.

LEMMA 4.1. (i) $S \leq S_T$, *and*
(ii) $A_T \leq (n-1)A$.

Proof. Inequality (i) follows directly from the triangle inequality. To prove (ii) consider a single move of the adversary when he moves a server from x to y. Consider the path P from x to y in T. If (u, v) is any edge on P, then $\|u, v\| \leq \|x, y\|$ because otherwise T would not be minimum. Since P has at most $n-1$ edges, $\|x, y\|_T \leq (n-1)\|x, y\|$, and (ii) follows. □

Since by Theorem 3.3 $S_T \leq kA_T$, we obtain the following.

THEOREM 4.2. *Algorithm* 2 *is $k(n-1)$-competitive in any n-point metric space.*

For some specific metric spaces this bound can yet be essentially improved. Denote by δ the *diameter* of T, that is, the maximum number of edges in a simple path of T. Then, by the same argument as above, we have that Algorithm 2 is in fact

$k\delta$-competitive. For example, if M is an $\sqrt{n} \times \sqrt{n}$ grid (with all weights equal to one) then it has a spanning tree of diameter $2\sqrt{n}$, so the competitiveness ratio is $2k\sqrt{n}$. If M is a hypercube (with all weights equal to one), then it has a spanning tree of diameter $\log n$, and then the competitiveness ratio is only $k \log n$.

5. Final remarks. Note that our algorithm works even when we relax our definition of a tree, allowing infinite trees. In fact, the tree may be dynamic, in the sense that we start with a single point, and each request consists of a request point and a path leading to this point from some already existing branch of the tree.

Example. A robot is confined to the space in a room above a certain track on the floor. The robot is always free to move vertically, but can only move horizontally along its track when it is resting on the floor. Let T be the set of positions of the robot, and $\|x, y\|$ is defined to be the total movement needed for the robot to get from position x to position y. Then T is a tree in this more general sense if the track is one-dimensional, in fact, if the track is any tree.

There is another property of our algorithm we would like to emphasize. In our approach we can view the servers as identical robots that move along the tree. By "identical" we mean that they have the same speed and execute the same program. Their program is simple: given the request site x move towards x unless you see another robot on your path to x (in which case, obviously, the other robot will reach x earlier so following it would be rather silly). We find it interesting that this somewhat chaotic behaviour, when robots actually compete between themselves to serve the request, leads to an effective algorithm.

Acknowledgments. Allan Borodin pointed out to us that our potential function can be expressed by the formula we currently use in the paper. The same formula was used in [3].

REFERENCES

[1] A. BORODIN, N. LINIAL, AND M. SAKS, *An optimal online algorithm for metrical task systems*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 373–382.

[2] M. CHROBAK, H. KARLOFF, T. PAYNE, AND S. VISHWANATHAN, *New results on server problems*, in Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 291–300.

[3] D. COPPERSMITH, P. G. DOYLE, P. RAGHAVAN, AND M. SNIR, *Random walks on weighted graphs and applications to online algorithms*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 369–378.

[4] A. FIAT, R. KARP, M. LUBY, L. MCGEOCH, D. SLEATOR, AND N. E. YOUNG, *Competitive paging algorithms*, Tech. Report CMU-CS-88-196, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988.

[5] S. IRANI AND R. RUBINFELD, *A competitive 2-server algorithm*, manuscript.

[6] A. KARLIN, M. MANASSE, L. RUDOLPH, AND D. SLEATOR, *Competitive snoopy caching*, Algorithmica, 3(1988), pp. 179–119.

[7] M. MANASSE, L. MCGEOCH, AND D. SLEATOR, *Competitive algorithms for server problems*, in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 322–333.

[8] L. MCGEOCH AND D. SLEATOR, *A strongly competitive randomized paging algorithm*, manuscript.

[9] P. RAGHAVAN AND M. SNIR, *Memory versus randomization in online algorithms*, in Proc. 16th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, Vol. 372, Springer-Verlag, Berlin, New York, 1989, pp. 687– 703.

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 149–155, February 1991
©1991 Society for Industrial and Applied Mathematics
009

THE COMPLEXITY OF THE RESIDUAL NODE CONNECTEDNESS RELIABILITY PROBLEM *

K. SUTNER†, A. SATYANARAYANA†, AND C. SUFFEL †

Abstract. This paper considers a probabilistic network in which the edges are perfectly reliable but the nodes fail with some known probabilities. The network is in an operational state if the surviving nodes induce a connected graph. The residual node connectedness reliability $R(G)$ of a network G is the probability that the graph induced by the surviving nodes is connected. This reliability measure is very different from the widely studied K-terminal network reliability measure. It is proven that the problem of computing the residual connectedness reliability is NP-hard by showing that the problem of counting the number of node induced connected subgraphs of a given graph is #**P**-complete. The problem remains #**P**-complete for split graphs as well as planar and bipartite graphs.

Key words. network reliability, planar graphs, bipartite graphs, #**P**-completeness

AMS(MOS) subject classifications. 68R10, 68Q15, 68R05

1. Introduction. A major issue in reliability theory is the determination of the reliability of a given system from the reliabilities of its components. System reliability includes a variety of network reliability problems that occur when the system is modelled as a graph or a digraph whose points or edges or both have an associated probability of being operational. Historically, network reliability has been concerned with the problem of determining the probability that there is a path of operational elements from a specified point to another point in the network. Recent developments in computer communication networks have led to an interest in more global measures and associated computational techniques. Consequently, various reliability measures have been defined in the literature. For example, one of the most commonly used performance measures is the *K-terminal reliability* of a graph. Suppose $G = \langle V, E \rangle$ is a graph and $K \subset V$ is a specified subset of V. Given that the elements (points and edges) of G may fail with known probabilities, the *K-terminal reliability* $R_K(G)$ of G is the probability that there is some subgraph H in G such that all elements of H are operational and all points of K lie in a single component of H.

If we restrict our attention to graphs G in which points do not fail but the edges fail independently of each other with equal probabilities ρ, then the K-terminal reliability of G can be expressed as a polynomial

$$R_K(G) = \sum_{i=1}^{|E|} S_i(G, K)\rho^{|E|-i}(1-\rho)^i,$$

where $S_i(G, K)$ is the number of subgraphs H of G such that H contains i edges and all points of K lie in a single component of H. Computing $R_K(G)$ in general is NP-hard, even for $|K| = 2$. This result was first proved by Valiant [7] by showing that the problem of computing the general term of the above polynomial is #**P**-complete. Subsequently, Provan [5] showed that even for planar graphs the computation of $R_K(G)$ is NP-hard. These results motivated the search for the classes of graphs G which admit polynomial-time algorithms for the computation of $R_K(G)$, for example, see [6], [2].

* Received by the editors October 23, 1989; accepted for publication (in revised form) July 9, 1990.

† Computer Science Department, Stevens Institute of Technology, Hoboken, New Jersey 07030.

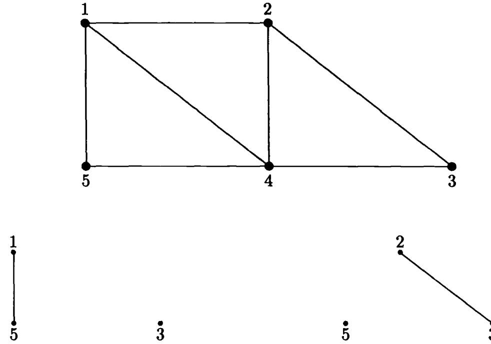


FIG. 1. *An example graph and its two 3-node failure states.*

A special case of the K-terminal problem is the following K-terminal node connectedness problem: In this model, edges do not fail, but the nodes that are not in a specified subset K do fail with known probabilities. The *K-terminal node connectedness reliability* of a graph G is then the probability that the surviving nodes of G induce a subgraph in which all nodes of K lie in a single component. This problem also was shown to be NP-hard for general graphs and it remains so even for chordal graphs and comparability graphs [1].

In this paper, we are concerned with the following reliability problem: The edges of the graph are perfectly reliable but the nodes fail independently of each other. The network is considered to be in an operational state if the surviving nodes induce a connected subgraph of G. The *residual node connectedness reliability* of a graph G, denoted $R(G)$, is the probability that the graph induced by the surviving nodes is nonempty and connected.

We first note that this problem is not a special case of the previous one; indeed it is very different from the K-terminal reliability problem. The K-terminal problem constitutes a hierarchical system while the residual connectedness problem does not. Specifically, let E be a finite set and $\mathcal{P}(E)$ be the power set of E.

A system (E, Ω) consists of E and a collection of operating states $\Omega \subset \mathcal{P}(E)$. A *hierarchical system* (E, Ω) is one where Ω is closed upward with respect to set inclusion, i.e., a superset of an operating state is again an operating state. We say that the system is operational if the collection of operating components is an operating state of the system. Assuming a probability distribution $\mathbf{Pr}$ on $\mathcal{P}(E)$, the reliability of the system is just $\mathbf{Pr}(\Omega)$. It is easily seen that the K-terminal model and its special case described above are hierarchical. The residual node connectedness model is not hierarchical since a supergraph of a connected graph may be disconnected, as illustrated in Fig. 1. Moreover, it seems that the computational aspects of the two problems are also very different.

For example, computing $R_K(K_n)$ for a complete graph on n nodes, in which

the edge failure probabilities are not necessarily equal, is clearly NP-hard. On the contrary, computing $R(K_n)$ is trivial because $R(K_n) = 1 - \prod \rho(i)$, where $\rho(i)$ is the failure probability of node i. Yet another example is the case where the given graph G is a tree. While $R_K(G) = \prod(1 - \rho(i))$, where $\rho(i)$ is the failure probability of edge i, computation of $R(G)$ requires a nontrivial, though linear time, algorithm; see [3] for details.

In this paper we show that computing the residual connectedness reliability is NP-hard. Indeed, we show that the problem remains hard even for split graphs and planar and bipartite graphs.

2. Complexity of the residual node connectedness problem. Let G be an undirected graph with e perfectly reliable edges and n nodes which fail independently and with equal probabilities ρ. Let $S_k(G)$ be the number of connected node induced subgraphs of G with k nodes. Then the residual node connectedness reliability $R(G)$ may be written as

$$R(G, \rho) = \sum_{k=0}^{n} S_k(G)\rho^{n-k}(1-\rho)^k.$$

We first show that the problem of computing $S(G) = \sum_{k=0}^{n} S_k(G)$ is #**P**-complete for split graphs G.

Since $R(G, \frac{1}{2}) = S(G)/2^n$ for $\rho = \frac{1}{2}$, it follows that computing $R(G)$ for split graphs is NP-hard. An undirected graph $G = \langle V, E\rangle$ is a *split graph* if there is a partition $V = I \cup C$ such that the nodes of I form an independent set of G while the nodes of C induce a clique in G.

THEOREM 2.1. *It is #**P**-complete to compute $S(G)$ for split graphs G.*

Proof. It is clear that computing $S(G)$ is in #**P**. For hardness we show that the problem of counting the number of satisfying truth assignments of a monotone boolean formula in 2-conjunctive normal form is polynomial-time Turing reducible to computing $S(G)$ for a suitably defined graph G. For the hardness of monotone 2-SAT, see [7].

Consider a boolean formula Φ in 2-CNF with variables $x_1, \cdots, x_r$ and clauses $c_1, \cdots, c_s$. We may safely assume that every variable occurs in at least one clause. For any $t \geq 1$ we now define a graph G^t associated with formula Φ as follows: G^t has vertices x_i, $i = 1, \cdots, r$, and c_j^τ, $j = 1, \cdots, s$, $\tau = 1, \cdots, t$ corresponding to the variables and clauses of Φ, respectively. Each clause is represented t times. There is an edge from x_i to c_j^τ (for all $\tau \leq t$) if and only if variable x_i occurs in clause c_j. Furthermore, there are edges that make $X = \{x_1, \cdots, x_r\}$ into a clique. Thus G^t is a split graph, see Fig. 2.

Let us define the weight of a truth assignment $\alpha : X \to \{0, 1\}$ to be

$$w(\alpha) := \text{ number of clauses of } \Phi \text{ satisfied by } \alpha.$$

Also let T_k be the number of satisfying truth assignments of weight k, $k = 0, \cdots, s$. Note that there is a natural class $\mathcal{C}_\alpha$ of connected subgraphs associated with every truth assignment α of weight at least 1. A connected subgraph C in $\mathcal{C}_\alpha$ has the form $C = X_\alpha \cup S$, where $X_\alpha := \{\, x \in X \mid \alpha(x) = 1 \,\}$ and S is an arbitrary subset of $\{\, c_j^\tau \mid \alpha \text{ satisfies clause } j, \tau = 1, \cdots, t \,\}$. For the sake of completeness define $\mathcal{C}_\emptyset := \{\, \{c_j^\tau\} \mid j = 1, \cdots, s, \tau = 1, \cdots, t \,\}$, where $\emptyset$ denotes the trivial truth assignment of weight 0. Observe that all these classes are disjoint. Furthermore, $\mathcal{C}_\alpha$ has cardinality

$(2^t)^{w(\alpha)}$ for all $\alpha \neq \emptyset$. It is easy to verify that every connected subgraph of G^t belongs to one of these classes.

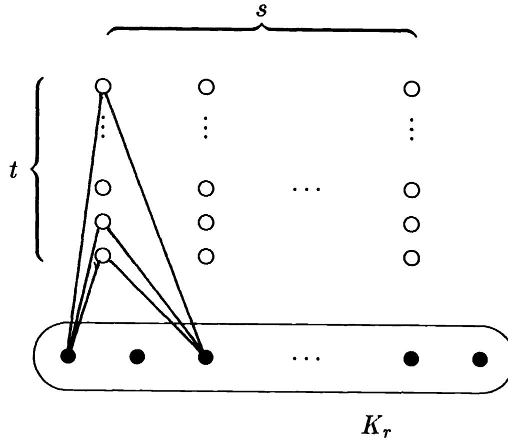


FIG. 2. *The split graph G^t constructed in the proof of Theorem* 2.1.

Consequently, we have

$$S(G^t) = st + \sum_{1 \le k \le s} T_k (2^t)^k.$$

Plainly, the right-hand side is essentially a polynomial of degree s with coefficients T_k. By choosing $s+1$ values of t we can thus compute the coefficients in polynomial-time, see [7]. But T_s is the number of satisfying truth assignments of Φ and we are done. □

In the following hardness argument for planar graphs we will use the fact that it is possible to protect certain vertices against deletion provided the total number of deleted vertices is small. More precisely, let $G = \langle V, E \rangle$ be an arbitrary graph on n points and $P \subset V$ a collection of nodes to be protected. Define a new graph $G(P)$ as follows: for every vertex v of G add $n+1$ new vertices $v_1, \cdots, v_{n+1}$ and edges $\{v, v_i\}$, $i = 1, \cdots, n+1$. Thus the new vertices v_i are endpoints in $G(P)$. Letting $p := |P|(n+1)$ the cardinality of $G(P)$ is $n' = n + p$.

Now define

$S_k(G; P) :=$ number of connected induced subgraphs of G on k points containing P.

We claim that for all $k \le n$

$$S_{n'-k}(G(P)) = \sum_{i \le k} \binom{p}{k-i} S_{n-i}(G; P). \tag{1}$$

To see this, first note that since $k \le n$ it is impossible to delete all the endpoints attached to any vertex in P. Hence the deletion of a vertex in P produces isolated vertices. But then any connected subset C of $G(P)$ of cardinality $n' - k$ contains all protected vertices and our claim follows. Note that $G(P)$ is planar and bipartite whenever G is.

Substituting $k = 0, \cdots, n$ in (1) we obtain a system of $n+1$ linear equations. Note that the system is in triangular form and each equation has leading coefficient

1. Hence we can compute the values $S_{n-i}(G;P)$, for $k = 0, \cdots, n$ from $S_{n'-k}(G(P))$ in polynomial time.

As in Lichtenstein [4], define a graph $gr(\Phi)$ associated with formula Φ in 3-CNF as follows. For each boolean variable x of Φ there is a vertex $v(x)$ in $gr(\Phi)$. As we will see below, $v(x)$ represents the variable x as well as the negated variable $\bar{x}$. Similarly, each clause c of Φ is represented by a vertex $v(c)$. There is an edge from $v(x)$ to $v(c)$ in $gr(\Phi)$ if and only if one of the literals x or $\bar{x}$ occurs in clause c. Furthermore, $gr(\Phi)$ contains an additional cycle through the vertices corresponding to variables.

The formula Φ is *planar* if and only if $gr(\Phi)$ is planar. It is shown in [4] that for every boolean formula Φ there exists a planar boolean formula Φ' which can be constructed from Φ in polynomial time that is satisfiable if and only if Φ is satisfiable. It is easy to verify that Lichtenstein's transformation is parsimonious, i.e., it preserves the number of satisfying truth assignments. For the hardness of 3-SAT, see [7]. Thus we have the following lemma.

LEMMA 2.2. *Counting the number of satisfying truth assignments of planar boolean formulae in* 3*-conjunctive normal form is* #**P***-complete.*

We will refer to this problem as P-3-SAT.

Let us define

$$\tilde{S}(G) := \sum_{k=0}^{n} S_k(G) \cdot 2^{kn}.$$

$\tilde{S}(G)$ will be used in the next theorem as a technical device to show that it is #**P**-complete to compute the sequence $S_0(G), \cdots, S_n(G)$.

THEOREM 2.3. *It is* #**P***-complete to compute* $\tilde{S}(G)$ *even if* G *is required to be planar and bipartite.*

Proof. It is clear that computing $\tilde{S}(G)$ is in #**P**. By Lemma 2.2 it suffices to show that P-3-SAT is polynomial-time Turing reducible to computing $\tilde{S}(G)$ where G is required to be planar and bipartite.

So assume Φ is a planar boolean formula Φ in 3-conjunctive normal form with, say, r variables and s clauses. Denote by X the set of variables of Φ and by C the set of clauses. For any variable x, let $\mu(x)$ be the number of occurrences of the literals x and $\bar{x}$ in Φ and set $m := \sum_{x \in X} \mu(x)$.

Now consider the planar graph $H = gr(\Phi)$. It is safe to assume that we have a planar embedding of H. In particular, we assume an appropriate cyclic ordering of the edges in H incident upon $v(x)$ for all the vertices $v(z)$ corresponding to boolean variables z in H.

We will modify H in a number of steps that preserve planarity and produce a new graph G. First we replace all the vertices $v(x)$, $x \in X$, by crossover boxes. We give a detailed description of one such box B, see also Fig. 3. Pick vertex $v(x) \in X$ and let $\mu = \mu(x)$.

The crossover box B is a "broken" wheel of size 4μ; more precisely, B has vertices

$$v(x_0), \cdots, v(x_{2\mu-1}), u, u_0, \cdots, u_{2\mu-1}$$

and edges

$$\{v(x_j), u\}, \{v(x_j), u_j\} \quad \text{and} \quad \{u_j, v(x_{j+1})\}$$

for all $j < 2\mu$ (here, as in the following, indices are supposed to be computed modulo some appropriate number). Thus u is the hub of the wheel, the vertices of the form

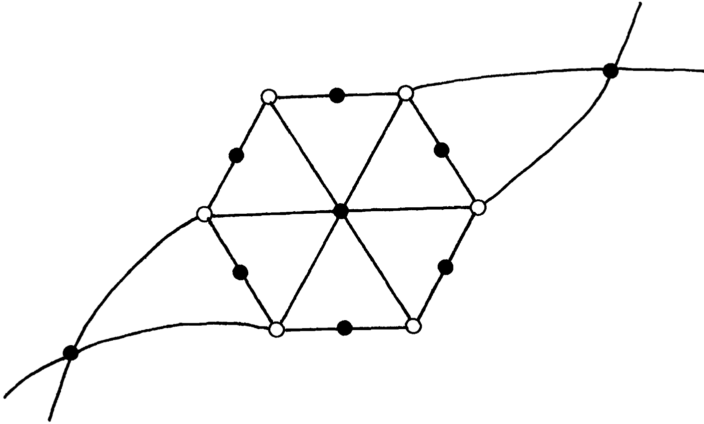


FIG. 3. *A crossover box in the graph representing formula Φ'. The corresponding variable has multiplicity 3. The full nodes are protected.*

$v(x_j)$ and u_j alternate on the perimeter, and the hub is connected to the $v(x_j)$ vertices only.

Next we replace the edges $\{v(x), v(c)\}$, $x \in X$, $c \in C$, in H by new edges $\{v(x_j), v(c)\}$ as prescribed by the planar embedding. We adopt the convention that j is chosen even whenever the occurrence of x in clause c is positive and odd otherwise. Furthermore, every vertex $v(x_j)$ is used at most once.

The last step is to connect the crossover boxes according to the cycle on X in H. To this end the old edges of the form $\{v(x), v(y)\}$, $x, y \in X$, are replaced by $\{v(x_j), u_{xy}\}$, $\{v(x_{j+1}), u_{xy}\}$, $\{u_{xy}, v(y_{j'})\}$, and $\{u_{xy}, v(y_{j'+1})\}$, where the u_{xy} are new vertices and j and j' are chosen according to the planar embedding.

A moment's thought shows that the resulting graph G on $n = s + 2r + 4m$ vertices is still planar. Furthermore, all cycles in G are necessarily of even length; hence G is in addition bipartite.

We now show how to interpret the changes in the graph in terms of the boolean formula. For each $x \in X$, introduce new boolean variables $x_0, \cdots, x_{2\mu(x)-1}$. Then replace each occurrence of x in clause c by x_{2j} whenever $\{v(x_{2j}), v(c)\}$ is an edge in G. Similarly, each occurrence of $\bar{x}$ is replaced by some x_{2j+1}, $0 \le j < \mu(x)$. Call the resulting boolean formula Φ'. Note that Φ' will in general fail to be planar.

Also define a formula

$$\Phi'' = \bigwedge_{\substack{x \in X \\ 0 \le j < 2\mu(x)}} x_j \vee x_{j+1}.$$

Lastly, let $\Psi = \Phi' \wedge \Phi''$. Thus all occurrences of boolean variables in Ψ are positive. As usual, we identify a truth assignment α of Ψ with the set of the variables satisfied by α, i.e., we identify α and $\{\, x_j \mid \alpha(x_j) = 1 \,\}$. Now set

$$T_i := |\{\, \alpha \mid \alpha \text{ a satisfying truth assignment of } \Psi, |\alpha| = 2m - i \,\}|.$$

Note that T_m is the number of satisfying truth assignments of the original formula Φ. Lastly, let P be the set of all vertices of G other than those of the form $v(x_j)$. We have the following claim.

Claim. The satisfying truth assignments of Ψ of cardinality $2m - i$ correspond bijectively to connected subgraphs S of G such that $P \subset S$ and S has cardinality $n - i$.

To see this, suppose S is a connected subset of G containing the protected vertices in P. S gives rise to a truth assignment α_S of Ψ:

$$\alpha_S(x_j) := \begin{cases} 1 & \text{if } x_j \in S, \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to check that α_S indeed satisfies Ψ. Conversely, every satisfying truth assignment α of Ψ can be translated into a corresponding connected subset S_α of G.

Computing the number of satisfying truth assignments of Φ thus amounts to computing T_m. By the claim, it suffices to determine $S_{n-i}(G;P)$, $i = 0, \cdots, m$, in order to compute the numbers $T_0, \cdots, T_m$. Lastly, by the remark following equation (1), the last problem in turn can be reduced to computing $\tilde{S}(G(P))$. Here G and therefore $G(P)$ are both planar and bipartite and we are done. □

The last result also shows that it is NP-hard to compute $R(G, \rho)$. To see this, note that $R(G, \rho) = \sum_{k=0}^{n} S_k(G)\rho^{n-k}(1-\rho)^k = \rho^n \sum_{k=0}^{n} S_k(G)((1-\rho)/\rho)^k$. Substituting $\tau = (1-\rho)/\rho$ in the last equation yields $R(G, \rho)/\rho^n = \sum_{k=0}^{n} S_k(G)\tau^k$. Since τ becomes arbitrarily large for ρ close to 0 one can use the techniques in Valiant [7] to retrieve the coefficients $S_0(G), \cdots, S_n(G)$ and therefore $\tilde{S}(G)$, given the values $R(G, \rho)/\rho^n$ for suitable choices of ρ. As we have just shown, the latter problem is #**P**-complete.

REFERENCES

[1] H. AboElFotoh and C. Colbourn, *Efficient algorithms for computing the reliability of permutation and interval graphs*, manuscript, 1989.

[2] C. Colbourn, *The combinatorics of network reliability*, Oxford University Press, London, U.K., 1987.

[3] C. Colbourn, A. Satyanarayana, C. Suffel, and K. Sutner, *Computing residual connectedness reliability for restricted networks*, Discrete Appl. Math., submitted.

[4] D. Lichtenstein, *Planar formulae and their uses*, SIAM J. Comput., 11 (1982), pp. 329–343.

[5] J. Provan, *The complexity of reliability computations in planar and acyclic graphs*, SIAM J. Comput., 15 (1986), pp. 695–702.

[6] A. Satyanarayana and R. Wood, *A linear time algorithm for computing K-terminal reliability in series-parallel networks*, SIAM J. Comput., 14 (1985), pp. 818–832.

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

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 156–183, February 1991
© 1991 Society for Industrial and Applied Mathematics
010

MORE NEARLY OPTIMAL ALGORITHMS FOR UNBOUNDED SEARCHING, PART I: THE FINITE CASE*

EDWARD M. REINGOLD† AND XIAOJUN SHEN‡

Abstract. Given a function $F : N^{+} \rightarrow \{\mathrm{X}, \mathrm{Y}\}$ with the property that if $F(n_0) = \mathrm{Y}$ then $F(n) = \mathrm{Y}$ for all $n > n_0$, the *unbounded search problem* is to use tests of the form "is $F(i) = \mathrm{X}$?" to determine the smallest n such that $F(n) = \mathrm{Y}$; the "cost" of a search algorithm is a function $c(n)$, the number of such tests used when the location of the first Y is n. A solution to this search problem specifies a prefix-free, binary encoding of the positive integers in which the cost $c(n)$ is the number of bits used to encode n. It is shown that the "ultimate algorithm," of Bentley and Yao [*Inform. Process. Lett.*, 5 (1976), pp. 82–87], which is within an additive $\Theta(\lg^* n)$ factor of a lower bound on the cost of this problem, is "far" from optimal in the sense that it is just the second in an infinite sequence of search algorithms, each of which is much closer to optimality than its predecessor. A corresponding sequence of lower bounds is also given, based on Kraft's inequality, each of which is much stronger than its predecessor. Diagonalizing over this sequence of search algorithms yields an algorithm, which is given explicitly in a Pascal-like notation, that is within an additive factor of $\alpha(n) + 2$ of the corresponding lower bound, where $\alpha(n)$ is a functional inverse of Ackermann's function—an *extremely* slowly growing function. For each search algorithm, the corresponding prefix-free, binary encoding of the integers is given, together with the decoding algorithm. Finally, algorithms/encodings are constructed that differ from the lower bounds by only negligible amounts even for the asymmetric case in which the cost of a Y answer and the cost of an X answer are not the same. In Part II it is shown how to continue the construction to get a transfinite sequence of dramatically better algorithms/encodings and lower bounds.

Key words. unbounded search, prefix-free codes, optimal algorithms, Ackermann's function, inverse Ackermann's function, Kraft's inequality

AMS(MOS) subject classifications. 68Q25, 68Q20, 94B45, 26A12

1. Introduction.

The *unbounded search problem*, introduced by Bentley and Yao [6], is to determine the location of the first Y value of a function F from the positive integers to the set {X, Y}, when F has the property that if $F(n_0) = \mathrm{Y}$, then $F(n) = \mathrm{Y}$ for all $n > n_0$. The "cost" of the search algorithm is a function $c(n)$ that specifies the number of tests "is $F(i) = \mathrm{X}$?" used when the location of the first Y is n. Bentley and Yao point out the equivalence of this problem and table lookup in an infinite ordered table, and they also show its connection to the construction of prefix-free, binary encodings of the integers in which the codeword for n has $c(n)$ bits. Knuth [14] contains an exposition of the problem of encodings and nomenclature for extremely large numbers.

Bentley and Yao describe an infinite sequence of increasingly better unbounded searching algorithms. They begin with a linear search that tests $F(1)$, $F(2)$, $F(3)$, $\cdots$ until $F(n) = \mathrm{Y}$. Second is a form of binary search in which the interval between

* Received by the editors April 30, 1989; accepted for publication (in revised form) June 2, 1990. Lest the reader think that the title is improperly hyphenated, we hasten to point out that the omission of hyphens is intentional: we could not choose among "More-Nearly-Optimal Algorithms for Unbounded Searching," "More-Nearly Optimal Algorithms for Unbounded Searching," or "More Nearly-Optimal Algorithms for Unbounded Searching;" each of these accurately describes the contents of this paper!

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ Computer Science and Telecommunications Program, University of Missouri-Kansas City, Kansas City, Missouri 64110. Part of this work was done at the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This author's research was partially supported by Office of Naval Research contract N00014-86K-0416 and by the East China Institute of Technology, Nanjing, China.

successive locations tested doubles until a Y is found; then bisection is used to find the first Y. Third is an algorithm in which successive tests occur at double powers of 2: $F(2^{2^0})$, $F(2^{2^1})$, $F(2^{2^2})$, $\cdots$ are tested until a Y is found; their second algorithm is then used to find the first Y. The sequence of algorithms continues, each algorithm using a higher number of exponentiated twos. Finally, they diagonalize over this sequence of algorithms to obtain what they call the "ultimate algorithm" which tests $F(1)$, $F(2^1)$, $F(2^{2^1})$, $\cdots$ until a Y is found; the height of the tower of twos at which the first Y is encountered determines how the search will continue. We will not give a further description here of their ultimate algorithm because it will be described, in passing, in the next section.

The cost of Bentley and Yao's ultimate algorithm is

$$c(n) = 4 + L^*(n) + \sum_{i=1}^{L^*(n)-1} L^i(n),$$

where $L^i(n)$ is similar to $\lfloor \lg^{(i)} n \rfloor$ and $L^*(n)$ is similar to $\lg^* n$. Specifically, they define

$$l^0(n) = n$$

and, for $i \geq 0$,

$$l^{i+1}(n) = \lfloor \lg l^i(n) \rfloor + 1;$$

then

$$L^i(n) = l^i(n) - 1,$$

while

$$L^*(n) = \text{smallest } i \text{ such that } L^i(n) = 1.$$

Bentley and Yao prove that their ultimate algorithm is within an additive factor of $\Theta(\lg^* n)$ of being optimal.

Approaching the problem from the point of view of constructing prefix-free, binary encodings of the positive integers,[1] Elias [7] and Lebenshtein [16] give constructions that are reminiscent of the encoding that corresponds to Bentley and Yao's ultimate algorithm: the encoding of n is $b_1 b_2 \cdots b_{c(n)}$, where $b_i = 1$ if and only if the ith evaluation of F is X. Elias comments that this scheme is "not quite ultimate," but that improvements require that n be "much larger than Eddington's estimate of the number of protons and electrons in the universe!"

In spite of the closeness to optimality of the ultimate algorithm or encoding, some attention has been paid to improvements. Stout [24] offers some very slight improvement; Knuth [14] outlines some significant (relatively!) improvements in the encodings and the lower bounds, and suggests still further possibilities. We outline Knuth's improvements in the next section, so we will not delve into them here. Beigel [3] gives a nonconstructive improvement on the algorithm and the lower bound, and remarks that an algorithm exists that is "within" $g(n)$ of being optimal, for any $g(n)$

[1] Such encodings, incidentally, are useful for assigning priorities to subtasks in a parallel problem-solving system; see [12] or [23].

that is unbounded and computable; we discuss this notion and its relation to our results later in this paper. Bentley and Yao [6] mention possible improvements in the lower bounds that deserve some comment here.

Lower bounds for the problem derive from *Kraft's inequality* [15] (see [6] or [14]): if $c(n)$ is the cost of an algorithm/encoding, then $\sum_{n=1}^{\infty} 2^{-c(n)} \leq 1$. If one can show, therefore, that $\sum_{n=0}^{\infty} 2^{-f(n)}$ diverges, then $c(n) > f(n)$ for infinitely many n, so that

$$\limsup_{n \to \infty} \frac{c(n)}{f(n)} \geq 1.$$

Bentley and Yao use this argument to obtain their lower bound. They go on to cite an unpublished result of Chung and Graham that

$$\sum_{n=1}^{\infty} \frac{1}{n(\lg n)(\lg\lg n)(\lg\lg\lg n)\cdots(\lg^{(\lg^* n)} n)(\lg^* n)}$$

diverges. If true, this would indeed yield a lower bound better than Bentley and Yao's; however, the sum converges, as can be seen by the comparison test and the fact that

$$\sum_{n=1}^{\infty} \frac{1}{n(\lg n)(\lg\lg n)(\lg\lg\lg n)\cdots(\lg^{(\lg^* n)} n)}$$

converges to approximately 2.865064. See inequality (B26) of [17] and [21, Appendix A]; [3, Lem. 21] also shows convergence, as does the integral test with an argument parallel to [2].

Both Knuth [14] and Beigel [3] present partial converses to Kraft's inequality. Knuth shows that if $c(1), c(2), c(3), \cdots$ is a nondecreasing sequence of positive integers such that $\sum_{n=1}^{\infty} 2^{-c(n)} = 1$, then an irredundant, prefix-free, binary encoding of the integers can be constructed in which $c(n)$ is the number of bits in the representation of n. The existence of such an encoding implies the existence of a corresponding unbounded search algorithm of cost $c(n)$: the encoding corresponds to an infinite binary search tree on the positive integers formed by interpreting 0 as a left edge and 1 as a right edge [14]. This tree is an infinite description of an unbounded search algorithm. Beigel [3] shows that for a computable, positive-integer-valued, nondecreasing function $c(n)$, there exists an unbounded search algorithm of cost $c(n)$ if and only if $\sum_{n=1}^{\infty} 2^{-c(n)} \leq 1$; his proof is nonconstructive, however.

Raoult and Vuillemin [19] and Goldstein and Reingold [9] have extended some of the results outlined above to *unimodal search*, in which we seek the location of the change in sign of the first derivative of a real-valued function. (We can think of the unbounded search problem as seeking the location of the change in sign of a real-valued function.) Raoult and Vuillemin also give independent derivations of some of the results in [14].

In the present paper we elucidate the nature of "near-optimal" algorithms/encodings by giving stupendous (relatively!) improvements in both the algorithms/encodings and the lower bounds. We use Ackermann's function to construct an infinite sequence of algorithms/encodings, and a corresponding sequence of lower bounds, in which the zeroth algorithm is a form of linear search, the first algorithm is a form of binary search, the second algorithm is almost identical to Bentley and Yao's ultimate algorithm, and the improvement from the ith algorithm to the $(i+1)$st algorithm is equally dramatic for any i. Diagonalizing over the zeroth, first, second, and so

on gives us an algorithm that is within an additive factor of $\alpha(n) + 2$ of the corresponding lower bound; $\alpha(n)$ is the functional inverse of Ackermann's function, an *extremely* slowly growing function that was first used in the analysis of algorithms by Tarjan [25]. This diagonal algorithm has a relatively simple Pascal-like implementation in terms of Ackermann's function. In a subsequent paper [20], we show how to continue the construction to get a transfinite sequence of dramatically better algorithms/encodings and lower bounds. This closes the gap between the cost of the best algorithm and the best lower bound to a razor-sharp edge. Furthermore, we construct algorithms/encodings that differ from the lower bounds by only negligible amounts even for the asymmetric case in which the cost of a Y answer and the cost of an X answer are not the same.

For notational simplicity, it is convenient to insist that $F(0) = \text{X}$ and to state the unbounded search problem as that of finding the greatest integer $n \geq 0$ such that $F(n) = \text{X}$; in other words, instead of looking for the first Y, we look for the last X. If the last X occurs at position $n \geq 0$, the search for the last X corresponds to the encoding of the integer n. Thus we will consider encodings of the nonnegative integers, following [14] instead of [6], who consider the positive integers only. These two views of unbounded searching are clearly equivalent: our cost function $c(n)$ is defined for $n \geq 0$ as the number of tests used to determine the location of the last X, when that location is at n; that is, $c(n)$ is the cost of determining that the first Y is at $n + 1$. Correspondingly, Kraft's inequality becomes

$$\sum_{n=0}^{\infty} 2^{-c(n)} \leq 1. \tag{1}$$

2. The level-by-level construction.

DEFINITION. *Ackermann's function* [1] is defined as follows for $n \geq 1$ (using the form given in [10], but with the subscripts shifted by 1):

$$A_i(n) = \begin{cases} 2n & i = 0,\ n \geq 1, \\ A_{i-1}^{(n)}(1) & i \geq 1, \end{cases}$$

where $A_{i-1}^{(j)}(n) = A_{i-1}(A_{i-1}^{(j-1)}(n))$, $A_{i-1}^{(0)}(n) = n$. For convenience, we define $A_i(0) = 1$ for $i \geq 0$.

Thus for $n \geq 1$, $A_1(n) = 2^n$ and $A_2(n) = \mathit{tower}(n)$, an n-high tower of exponentiated twos. Some values of $A_i(n)$ are shown in Table 1.

DEFINITION. For $n \geq 0$, the *inverse Ackermann's function* is defined by

$$\alpha_i(n) = \begin{cases} \lfloor n/2 \rfloor & i = 0, \\ \text{least } j \text{ such that } \alpha_{i-1}^{(j)}(n) \leq 1 & i \geq 1, \end{cases}$$

where $\alpha_{i-1}^{(j)} = \alpha_{i-1}(\alpha_{i-1}^{(j-1)}(n))$, $\alpha_{i-1}^{(0)}(n) = n$.

Thus, for example, $\alpha_1(n) = \lfloor \lg n \rfloor$, for $n \geq 1$, and $\alpha_2(n) = \lg^* n$. Some values of $\alpha_i(n)$ are shown in Table 2. For $i \geq 0$, $\alpha_i(n)$ is the functional inverse of $A_i(n)$, because

$$\alpha_i(n) = A_i^{-1}(n) = \text{greatest } x \geq 1 \text{ such that } A_i(x) \leq n,$$

and hence $\alpha_i(A_i(n)) = n$. Since $\alpha_i(n)$ is many-to-one, it has no such inverse; however

$$A_i(\alpha_i(n)) = \text{least } x \geq 1 \text{ such that } \alpha_i(x) = \alpha_i(n).$$

TABLE 1

Ackermann's function; tower(n) is an n-high tower of exponentiated twos. Omitted values are far too large to be expressed conveniently in the table. The functions $A_i(n)$ are defined in §2 and $A(n)$ is defined in §3.

n	0	1	2	3	4	5	6	7	8
$A_0(n) = 2n, n \geq 1$	1	2	4	6	8	10	12	14	16
$A_1(n) = 2^n$	1	2	4	8	16	32	64	128	256
$A_2(n) = tower(n)$	1	2	4	16	2^{16}	$2^{2^{16}}$	$2^{2^{2^{16}}}$	$tower(7)$	$tower(8)$
$A_3(n)$	1	2	4	2^{16}	$tower(2^{16})$	$tower(tower(2^{16}))$			
$A_4(n)$	1	2	4	$tower(2^{16})$	$A_3(tower(2^{16}))$				
$A(n)$	1	2	4	2^{16}					

TABLE 2

Inverse Ackerman's function. The functions $\alpha_i(n)$ are defined in §2 and $\alpha(n)$ is defined in §3.

n	0	1	2	3	4	5	6	7	$8 \cdots$	15	$16 \cdots$	31	$32 \cdots$	63	$64 \cdots$	$2^{16}-1$	$2^{16} \cdots$	$2^{2^{16}}$
$\alpha_0(n) = \lfloor n/2 \rfloor$	0	0	1	1	2	2	3	3	$4 \cdots$	7	$8 \cdots$	15	$16 \cdots$	31	$32 \cdots$	$2^{15}-1$	$2^{15} \cdots$	$2^{2^{16}-1}$
$\alpha_1(n) = \lfloor \lg n \rfloor$	0	0	1	1	2	2	2	2	$3 \cdots$	3	$4 \cdots$	4	$5 \cdots$	5	$6 \cdots$	15	$16 \cdots$	2^{16}
$\alpha_2(n) = \lg^* n$	0	0	1	1	2	2	2	2	$2 \cdots$	2	$3 \cdots$	3	$3 \cdots$	3	$3 \cdots$	3	$4 \cdots$	5
$\alpha_3(n)$	0	0	1	1	2	2	2	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$3 \cdots$	3
$\alpha_4(n)$	0	0	1	1	2	2	2	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2
$\alpha(n)$	0	0	1	1	2	2	2	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$2 \cdots$	2	$3 \cdots$	3

As Table 1 indicates, the values of $A_i(n)$ grow rapidly. Specifically, $A_{i+1}(n)$ grows much faster than $A_i(n)$ as $n \to \infty$, for any i. Consequently, the inverse functions $\alpha_i(n)$ grow slowly, with $\alpha_{i+1}(n)$ growing much slower than $\alpha_i(n)$ as $n \to \infty$, for any i.

DEFINITION. The *length function* $L_i(n)$ is defined by

$$L_0(n) = \begin{cases} 0 & n \leq 1, \\ 1 & n > 1, \end{cases}$$

and by

$$L_i(n) = \begin{cases} 0 & n \leq 1, \\ L_{i-1}(n) + L_i(\alpha_{i-1}(n)) & n > 1, \end{cases}$$

for $i \geq 1$.

Thus, for example, $L_1(n) = \lfloor \lg n \rfloor$ and $9L_2(n) = \lfloor \lg n \rfloor + \lfloor \lg \lg n \rfloor + \lfloor \lg \lg \lg n \rfloor + \cdots + \lfloor \lg^{(\lg^* n)} n \rfloor$, interpreting $\lfloor \lg 0 \rfloor$ as 0. In general, we have

$$L_i(n) = \sum_{j=0}^{\alpha_i(n)-1} L_{i-1}(\alpha_{i-1}^{(j)}(n)),$$

and

$$L_i(n) = \sum_{j_t \geq 0, 1 \leq t < i} \lfloor \lg \alpha_1^{(j_1)}(\alpha_2^{(j_2)}(\cdots(\alpha_{i-1}^{(j_{i-1})}(n))\cdots)) \rfloor.$$

Some values of $L_i(n)$ are shown in Table 3. In the notation of Knuth [14], his λn is our $\alpha_1(n)$, his $\lambda^* n$ is our $\alpha_2(n) + 1$, and his Λn is our $L_2(n)$.

TABLE 3
The length functions $L_i(n)$ and $L(n)$. The $L_i(n)$ are defined in §2 and $L(n)$ is defined in §3.

n	0	1	2	3	4	5	6	7	$8 \cdots 15$	$16 \cdots 31$	$32 \cdots 63$	$64 \cdots 2^{16}-1$	$2^{16} \cdots 2^{2^{16}}$
$L_0(n)$	0	0	1	1	1	1	1	1	$1 \cdots 1$	$1 \cdots 1$	$1 \cdots 1$	$1 \cdots 1$	$1 \cdots 1$
$L_1(n)$	0	0	1	1	2	2	2	2	$3 \cdots 3$	$4 \cdots 4$	$5 \cdots 5$	$6 \cdots 15$	$16 \cdots 2^{16}$
$L_2(n)$	0	0	1	1	3	3	3	3	$4 \cdots 4$	$7 \cdots 7$	$8 \cdots 8$	$9 \cdots 19$	$23 \cdots 2^{16}+23$
$L_3(n)$	0	0	1	1	4	4	4	4	$5 \cdots 5$	$8 \cdots 8$	$9 \cdots 9$	$10 \cdots 20$	$27 \cdots 2^{16}+27$
$L_4(n)$	0	0	1	1	5	5	5	5	$6 \cdots 6$	$9 \cdots 9$	$10 \cdots 10$	$11 \cdots 21$	$28 \cdots 2^{16}+28$
$L(n)$	0	0	1	1	4	4	4	4	$5 \cdots 5$	$8 \cdots 8$	$9 \cdots 9$	$10 \cdots 20$	$28 \cdots 2^{16}+28$

LEMMA. *For all $i \geq 0$, $m \geq 0$,*

$$\sum_{n \geq 1, \alpha_i(n)=m} 2^{-L_i(n)} = 1.$$

Proof. The proof is by double induction on i and m. For the basis of the induction, we have, for all $i \geq 0$,

$$\sum_{n \geq 1, \alpha_i(n)=0} 2^{-L_i(n)} = 2^{-L_i(1)} = 2^{-0} = 1$$

since $\alpha_i(n) = 0$ only for $n \leq 1$. We also have, for all $m \geq 1$,

$$\sum_{n \geq 1, \alpha_0(n)=m} 2^{-L_0(n)} = \frac{1}{2} + \frac{1}{2} = 1,$$

by the definition of $L_0(n)$.

Now, suppose the result is true for all m for $0 \leq i < i'$ and for i' when $0 \leq m < m'$. Observe that for $i' > 0$, by the definition of $\alpha_{i'}(n)$, we have $\alpha_{i'}(\alpha_{i'-1}(n)) = \alpha_{i'}(n) - 1$, so that by the definition of $L_i(n)$,

$$\begin{aligned}\sum_{n \geq 1, \alpha_{i'}(n)=m'} 2^{-L_{i'}(n)} &= \sum_{n \geq 1, \alpha_{i'}(\alpha_{i'-1}(n))=m'-1} 2^{-L_{i'-1}(n)-L_{i'}(\alpha_{i'-1}(n))} \\ &= \sum_{k \geq 1, \alpha_{i'}(k)=m'-1} 2^{-L_{i'}(k)} \sum_{n \geq 1, \alpha_{i'-1}(n)=k} 2^{-L_{i'-1}(n)},\end{aligned}$$

by substituting k for $\alpha_{i'-1}(n)$ and rewriting. But by induction the inner summation is 1, giving

$$\begin{aligned} &= \sum_{k \geq 1, \alpha_{i'}(k)=m'-1} 2^{-L_{i'}(k)} \\ &= 1,\end{aligned}$$

again by induction. □

Remark. The lemma can be rephrased as: for all $i \geq 0$, $m \geq 0$,

$$\sum_{n=A_i(m)}^{A_i(m+1)-1} 2^{-L_i(n)} = 1.$$

Knuth [14] proved the lemma for the special case $i = 2$ and noted that the proof technique works as well for the function $\Lambda n + \Lambda\lambda^* n + \Lambda\lambda^*\lambda^* n + \cdots + \Lambda(\lambda^*)^m n + \lambda(\lambda^*)^{m+1} n$, a truncated version of our $L_3(n)$. Raoult and Vuillemin [19] also allude to a truncated version of the lemma for $i = 3$.

COROLLARY 2.1. *For all $i \geq 0$, $m \geq 1$,*

$$\alpha_i(m) \leq \sum_{n=1}^{m-1} 2^{-L_i(n)} < \alpha_i(m) + 1.$$

COROLLARY 2.2. *For all $i \geq 0$, $\sum_{n=1}^{\infty} 2^{-L_i(n)}$ diverges.*

The functions L_i also provide, unexpectedly, a solution to a problem in [18]. We have the following corollary.

COROLLARY 2.3. *If $f(x) = x2^{-L_i(\lg x)}$ and*

$$M(n) = \max_{1\leq k<n} (M(k) + M(n-k) + \min(f(k), f(n-k))),$$

then $M(n) = \Theta(n\alpha_i(\lg n))$.

COROLLARY 2.4. *For all $i \geq 0$, $L_i(n)$ is a lower bound on the unbounded searching problem of Bentley and Yao* [6] *and on prefix-free, binary encodings of the nonnegative integers* [7] (*see* [14]), *in the sense that if an unbounded search algorithm uses $c(n)$ probes to locate n (or, equivalently, a prefix-free code uses $c(n)$ bits to encode n) then $c(n) > L_i(n)$ for infinitely many n.*

Proof. This follows from Corollary 2. by Kraft's inequality (1). □

THEOREM 2.5. *For all $i \geq 0$, there is an unbounded search algorithm in which the cost of finding n is*

$$L_i(n) + \alpha_i(n) + \begin{cases} 2 & n \leq 1, \\ 1 & n > 1. \end{cases}$$

Proof. We proceed recursively, constructing the level i search algorithm by applying the algorithm at lower levels. To make the recursion work, we introduce the notion of the location of the last X, *relative to a function f*: we say that u is the location of the last X relative to f if u satisfies

$$f(u) \leq \text{location of the last X} < f(u+1).$$

The critical observation to make is that if we are given a search algorithm $\mathcal{S}$ that finds n, the location of the last X, in $c(n)$ tests, then for any monotone, computable, increasing function f we can use $\mathcal{S}$ to find the location of the last X relative to f in only $c(f^{-1}(n))$ tests, where $f^{-1}(n)$ is the functional inverse of $f(n)$ defined by

$$f^{-1}(n) = \text{greatest } x \geq 1 \text{ such that } f(x) \leq n.$$

All we need do is replace each test "$F(i) = \text{X}$?" in $\mathcal{S}$ with the test "$F(f(i)) = \text{X}$?".

Suppose we are given u, the location of the last X relative to the function $f(A_l(n))$. The search can find the location of the last X relative to f by proceeding as follows. If $u = 0$, $l \geq 1$, we know that

$$f(1) = f(A_l(0)) \leq \text{location of the last X} < f(A_l(1)) = f(2),$$

since $A_l(0) = 1$ and $A_l(1) = 2$ for all l; thus the location of the last X relative to f must be 1. (The case $l = u = 0$ will never occur, as will be apparent.) On the other hand, if $u > 0$ and $l = 0$, we know that

$$f(2u) = f(A_0(u)) \leq \text{location of the last X} < f(A_0(u+1)) = f(2u+2),$$

so the single test "$F(f(2u+1)) = X$?" resolves whether the location of the last X relative to f is $2u$ or $2u+1$.

The most difficult case is $u > 0$ and $l > 0$. Here we know that u is the location of the last X relative to $f(A_l(n))$, so that

$$f(A_l(u)) \leq \text{location of the last X} < f(A_l(u+1)). \tag{2}$$

We can proceed recursively because

$$f(A_l(n)) = f(A_{l-1}(A_l(n-1))).$$

Letting $\hat{f}(n) = f(A_{l-1}(n))$, inequality (2) becomes

$$\hat{f}(A_l(u-1)) \leq \text{location of the last X} < \hat{f}(A_l(u));$$

that is, we know that the location of the last X relative to $\hat{f}(A_l(n))$ is $u-1$. This allows us to find the location of the last X: we recursively find the location of the last X relative to $\hat{f}(n) = f(A_{l-1}(n))$ and then we use *that* value to find (recursively) the location of the last X relative to f.

The structure of the recursion is as follows. Given a level l, a function f, and the location of the last X relative to $f(A_l(n))$, we can find the location of the last X relative to f. If such a search is called *LevelSearch*, an abuse of Pascal-like notation allows us to write

```
if u = 0 then {f(1) ≤ location of the last X < f(2)}
    return(1)
else if l = 0 then {f(2u) ≤ location of the last X < f(2u + 2)}
    if F(f(2u + 1)) = X then return(2u + 1) else return(2u)
else {go down a level}
    return(LevelSearch(l − 1, LevelSearch(l, u − 1, f(A_{l−1})), f))
```

to specify the recursion. Getting the process started means only determining k, the location of the last X relative to $A_{level}(n)$ for the desired *level*, and then applying the above outlined *LevelSearch* to find the location of the last X relative to the identity function. Finding the location of the last X relative to $A_{level}(n)$ is easily done by a **while** loop that successively tests $F(A_{level}(1))$, $F(A_{level}(2))$, $F(A_{level}(3))$, $\cdots$.

The complete procedure is shown in Algorithm 1. The correctness of this procedure follows by mathematical induction, as outlined in the preceding paragraphs. Notice the intense similarity between the function *LevelSearch* and a recursively written function that computes $A_l(n)$ (see, for example, [22, Fig. 5.6, p. 107]).

It remains to analyze the number of tests made in a search. When n, the location of the last X, is 0 or 1, one test is made in the **while** loop and one is made afterward, since $k = 0$; thus two tests are made. For $n > 1$, the **while** loop at the beginning of *search* uses $\alpha_{level}(n) + 1$ tests, because $A_{level}(\alpha_{level}(n) + 1)$ is the first value encountered that is larger than n. All the remaining tests are done in *LevelSearch*. Let $s_{level}(n)$ be the number of tests done in *LevelSearch*. We see that

$$s_{level}(1) = 0,$$

ALGORITHM 1
Level-by-level searching, in pseudo-Pascal.

```
function search(
    level: integer; { level of the search }
    F: function(integer): [X, Y] { if F(n_0) = Y then F(n) = Y for all n ≥ n_0 }
    ): integer; { the location of the last X of F }

var
    k: integer; { location of the last X relative to A_level;
                  that is, A_level(k) ≤ location < A_level(k + 1)}

function LevelSearch(
    l: integer; { level of the search }
    u: integer; { location of the last X, relative to f(A_l) }
    f: function(integer): integer
    ): integer; { location of the last X, relative to f; that is, i such that
                  f(i) ≤ location < f(i + 1) }

begin {LevelSearch}

    { find the location of the last X relative to f, given that its location relative
      to f(A_l) is u; that is, u satisfies f(A_l(u)) ≤ location < f(A_l(u + 1)) }

    if u = 0 then { f(1) ≤ location of the last X < f(2) }
        return(1)
    else if l = 0 then { f(2u) ≤ location of the last X < f(2u + 2) }
        if F(f(2u + 1)) = X then return(2u + 1) else return(2u)
    else { go down a level }
        return(LevelSearch(l − 1, LevelSearch(l, u − 1, f(A_{l−1})), f))

end; { LevelSearch }

begin { search }

    k := 0;
    while F(A_level(k + 1)) = X do
        k := k+1;

    { k > 0 is the location of the last X, relative to A_level;
      that is, A_level(k) ≤ location < A_level(k + 1) }

    if k = 0 then { the location of the last X is either 0 or 1 }
        if F(1) = X then return(1) else return(0)
    else
        return(LevelSearch(level, k, f(x) = x))

end; { search }
```

since no tests are made when $u = 0$, which is the case when $n = 1$. When $u > 0$, $level = 0$, exactly one test is made, so that

$$s_0(n) = 1.$$

When $u > 0$, $level > 0$, exactly $s_{level}(A_{level-1}^{-1}(n)) = s_{level}(\alpha_{level-1}(n))$ tests are made in the inner recursive call, while $s_{level-1}(n)$ tests are made in the outer recursive call; thus

$$s_{level}(n) = s_{level-1}(n) + s_{level}(\alpha_{level-1}(n)),$$

and we see that $s_{level}(n)$ is, therefore, identical to $L_{level}(n)$. The total number of tests made when $n > 1$ is the location of the last X is thus $L_{level}(n) + \alpha_{level}(n) + 1$. □

Several comments are in order. First, *search*(0, F) is a form of linear search in which alternate values are probed until a Y is found. Similarly, *search*(1, F) is a form of binary search in which the interval between successive probes doubles until a Y is found. Finally, *search*(2, F) is essentially Bentley and Yao's ultimate search [6]. Each of these search strategies is better than the previous one, once the location of the last X becomes sufficiently large. Specifically, for all i there exists an n_i such that *search*$(i+1, F)$ is faster than *search*(i, F) for all $n > n_i$; this is much stronger than saying that *search*$(i+1, F)$ "dominates" *search*(i, F) in the sense of Knuth [14].

Corollary 2.4 tells us that the search procedure in Algorithm 1 is within an additive factor $\alpha_{level}(n) + 2$ of being optimal. For $level = 2$, this is equivalent to Bentley and Yao's result [6]. For each $level > 2$, this is an enormous improvement because $\alpha_{i+1}(n)$ is much more slowly growing than $\alpha_i(n)$ as $n \to \infty$.

Each of these search procedures corresponds to a prefix-free, binary encoding of the nonnegative integers: the corresponding encoding for n is the sequence of bits obtained by taking the sequence $F(x_0)$, $F(x_1)$, $\cdots$ used in determining that the location of the last X is at n, and replacing each X by a 1 and each Y by a 0 (see [14], for example).

To construct these encodings explicitly, let

$$B_0(n) = \begin{cases} \text{empty string} & n = 1, \\ n \bmod 2 & n > 1, \end{cases}$$

and for $i \geq 1$, let

$$B_i(n) = \begin{cases} \text{empty string} & n = 1, \\ B_i(\alpha_{i-1}(n))B_{i-1}(n) & n > 1. \end{cases}$$

$B_1(n)$ is the binary representation of n with the leading 1-bit deleted. $B_2(n)$ is a version of an encoding given in [8]. The length of $B_i(n)$ is $L_i(n)$.

For $i \geq 0$ define

$$C_i(n) = \begin{cases} 00 & n = 0, \\ 01 & n = 1, \\ 1^{\alpha_i(n)}0B_i(n) & n \geq 2. \end{cases}$$

The length of $C_i(n)$ is thus

$$\text{length}(C_i(n)) = \begin{cases} 2 & n \leq 1, \\ L_i(n) + \alpha_i(n) + 1 & n > 1. \end{cases}$$

The encoding $C_i(n)$ precisely reflects the sequence of tests made in $search(i, F)$ when $F(n)$ is the last X.

Knuth [14] gives the encoding C_1 and a variation of C_2 that he calls R; specifically, he defines R to be

$$R(n) = \begin{cases} 0 & n = 0, \\ 1^{\alpha_2(n)+1} 0 B_2(n) & n \geq 1, \end{cases}$$

and he gives an algorithm to decode $R(n)$ into n. Knuth goes on to give a sequence of encodings R^{m+1}, $m \geq 1$, such that R^{m+1} is better than R^m; each of the encodings in this sequence is inferior to C_3, however. Raoult and Vuillemin [19] give similar, though less explicit, constructions like R^{m+1}.

THEOREM 2.6. *For all $i \geq 0$, C_i is a uniquely decipherable, irredundant, lexicographic, prefix-free code for the nonnegative integers in which the number of bits used to represent n is*

$$L_i(n) + \alpha_i(n) + \begin{cases} 2 & n \leq 1, \\ 1 & n > 1. \end{cases}$$

Proof. The lengths of the codewords are obvious by construction.

Unique decipherability follows by giving a decoding procedure and proving it correct. Such a procedure, given in Algorithm 2, is strikingly similar in form to the search procedure in Algorithm 1. The proof of its correctness is an inductive argument that mirrors the recursive nature of the procedure *DecodeB*. Specifically, the correctness of the procedure *DecodeC* follows from verifying that if $prefix = \alpha_l(n)$ and the bit sequence b_{k+1}, b_{k+2}, $\cdots$ begins with the bits of $B_l(n)$, then *DecodeB* returns n as its value. If $l = 0$, the value of $prefix$ is $\alpha_0(n) = \lfloor n/2 \rfloor$ and $b_{k+1} = n \bmod 2$, so the correct value 1 is returned if $prefix = 0$ and the correct value $n = 2 \times prefix + b_{k+1}$ is returned if $prefix > 0$. If $l > 0$ and $prefix = 0$, then

$$1 = A_l(0) \leq n < A_l(1) = 2,$$

so the correct value 1 is returned. Continuing inductively, suppose *DecodeB* is correct for all n when $0 \leq l < l'$ and is correct for all n satisfying $\alpha_{l'}(n) < prefix$ for level l'. Then the inner recursive call correctly returns $\alpha_{l'-1}(n)$ and the outer recursive call correctly returns n. A similar induction shows that C_i is lexicographic.

The structure of the decoding algorithm implies that C_i is prefix-free: the algorithm scans the bits from left to right without ever having to back up or look ahead.

To show that the codes C_i are irredundant, it suffices to verify that

$$\sum_{n=0}^{\infty} 2^{-\operatorname{length}(C_i(n))} = 1$$

(see, for example, [14]). We have

$$\begin{aligned} \sum_{n=0}^{\infty} 2^{-\operatorname{length}(C_i(n))} &= \frac{1}{4} + \frac{1}{4} + \sum_{n=2}^{\infty} 2^{-L_i(n)-\alpha_i(n)-1} \\ &= \frac{1}{2} + \frac{1}{2} \sum_{n=2}^{\infty} 2^{-L_i(n)-\alpha_i(n)}. \end{aligned}$$

ALGORITHM 2
Decoding the codes in the level-by-level construction, in pseudo-Pascal.

```
function DecodeC(
    level: integer; { level of the code }
    b: BitString { the codeword to be decoded }
    ): integer; { the decoded value }

var
    k: integer; { index of last bit of b processed }
    p: integer; { value of prefix part of b }

function DecodeB(
    l: integer; { level of the code }
    prefix: integer { prefix value }
    ): integer; { decoded value }

begin { DecodeB }

    if prefix = 0 then { the empty string encodes 1 }
        return(1)
    else if l = 0 then begin { value encoded is twice the prefix plus the last bit }
        k := k + 1;
        return(2 * prefix + b_k)
        end
    else { decode the next prefix and use it to decode the remaining bits }
        return(DecodeB(l − 1, DecodeB(l, prefix − 1)))

end; { DecodeB }

begin { DecodeC }

    k := 0;
    while b_{k+1} = 1 do
        k := k+1;

    { b_1 = ··· = b_k = 1 and b_{k+1} = 0 }

    p := k;
    k := k + 1; { skip the 0 }
    if p = 0 then
        if b_2 = 1 then return(1) else return(0)
    else
        return(DecodeB(level, p))

end; { DecodeC }
```

$$= \frac{1}{2} + \frac{1}{2} \sum_{m=1}^{\infty} \sum_{n \geq 1, \alpha_i(n) = m} 2^{-L_i(n) - \alpha_i(n)}$$
$$= \frac{1}{2} + \frac{1}{2} \sum_{m=1}^{\infty} 2^{-m} \sum_{n \geq 1, \alpha_i(n) = m} 2^{-L_i(n)}$$
$$= \frac{1}{2} + \frac{1}{2} \sum_{m=1}^{\infty} 2^{-m}$$

by the lemma, and hence

$$= \frac{1}{2} + \frac{1}{2} 1 = 1. \qquad \square$$

3. Diagonalization.

DEFINITION. The *diagonal Ackermann's function* is $A(n) = A_n(n)$.

DEFINITION. The *inverse diagonal Ackermann's function* is

$$\begin{aligned} \alpha(n) &= \text{greatest } j \text{ such that } \alpha_j(n) \geq j \\ &= \text{greatest } j \text{ such that } A_j(j) \leq n. \end{aligned}$$

$\alpha(n) = A^{-1}(n)$, the functional inverse of A, in the sense that $\alpha(A(n)) = n$. Since α is many-to-one, it has no inverse, however,

$$A(\alpha(n)) = \text{least } x \geq 1 \text{ such that } \alpha(x) = \alpha(n).$$

As $n \to \infty$, $A(n)$ grows enormously fast, faster than any $A_i(n)$. In fact, $A(n)$ grows faster than any primitive recursive function of n. Conversely, as $n \to \infty$, the inverse function $\alpha(n)$ grows enormously slower than any $\alpha_i(n)$.

DEFINITION. The *diagonal length function* is defined by

$$L(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ L_{\alpha(n)}(n) + \alpha_{\alpha(n)}(n) - \alpha(n) & \text{if } \alpha_{\alpha(n)}(n) = \alpha_{\alpha(n)}(A(\alpha(n)+1) - 1), \\ L_{\alpha(n)}(n) + \alpha_{\alpha(n)}(n) - \alpha(n) + 1 & \text{otherwise.} \end{cases}$$

THEOREM 3.1. *For all* $m \geq 0$,

$$\sum_{n \geq 1, \alpha(n) = m} 2^{-L(n)} = 1.$$

Proof. For $m = 0$,

$$\sum_{n \geq 1, \alpha(n) = 0} 2^{-L(n)} = 2^{-L(1)} = 1.$$

For $m \geq 1$,

$$\sum_{n \geq 1, \alpha(n) = m} 2^{-L(n)} = \sum_{i=m}^{\alpha_m(A(m+1)-1)} \sum_{n \geq 1, \alpha_m(n) = i} 2^{-L(n)}$$

$$
\begin{aligned}
&= \sum_{i=m}^{\alpha_m(A(m+1)-1)-1} \sum_{n\geq 1, \alpha_m(n)=i} 2^{-L_m(n)-\alpha_m(n)+\alpha(n)-1} \\
&\quad + \sum_{n\geq 1, \alpha_m(n)=\alpha_m(A(m+1)-1)} 2^{-L_m(n)-\alpha_m(n)+\alpha(n)} \\
&= \sum_{i=m}^{\alpha_m(A(m+1)-1)-1} \sum_{n\geq 1, \alpha_m(n)=i} 2^{-L_m(n)-i+m-1} \\
&\quad + \sum_{n\geq 1, \alpha_m(n)=\alpha_m(A(m+1)-1)} 2^{-L_m(n)-\alpha_m(A(m+1)-1)+m} \\
&= \sum_{i=m}^{\alpha_m(A(m+1)-1)-1} 2^{-i+m-1} \sum_{n\geq 1, \alpha_m(n)=i} 2^{-L_m(n)} \\
&\quad + 2^{-\alpha_m(A(m+1)-1)+m} \sum_{n\geq 1, \alpha_m(n)=\alpha_m(A(m+1)-1)} 2^{-L_m(n)} \\
&= \sum_{i=m}^{\alpha_m(A(m+1)-1)-1} 2^{-i+m-1} + 2^{-\alpha_m(A(m+1)-1)+m},
\end{aligned}
$$

by the lemma, and hence

$$
\begin{aligned}
&= \left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\alpha_m(A(m+1)-1)-m}}\right) + \frac{1}{2^{\alpha_m(A(m+1)-1)-m}} \\
&= 1.
\end{aligned}
$$

□

Remark. The theorem can be rephrased as: for all $m \geq 1$,

$$\sum_{n=A(m)}^{A(m+1)-1} 2^{-L(n)} = 1.$$

COROLLARY 3.2. *For all* $m \geq 1$,

$$\alpha(m) \leq \sum_{n=1}^{m-1} 2^{-L(n)} < \alpha(m) + 1.$$

COROLLARY 3.3. $\sum_{n=1}^{\infty} 2^{-L(n)}$ *diverges.*

Again, we have the solution to a problem in [18] in the following corollary.

COROLLARY 3.4. *If* $f(x) = x2^{-L(\lg x)}$ *and*

$$M(n) = \max_{1\leq k<n} (M(k) + M(n-k) + \min(f(k), f(n-k))),$$

then $M(n) = \Theta(n\alpha(\lg n))$.

COROLLARY 3.5. $L(n)$ *is a lower bound on the unbounded searching problem of Bentley and Yao* [6] *and on prefix-free, binary encodings of the nonnegative integers* [7] *(see* [14]*), in the sense that if an unbounded search algorithm uses* $c(n)$ *probes to locate* n *(or, equivalently, a prefix-free code uses* $c(n)$ *bits to encode* n*) then* $c(n) > L(n)$ *for infinitely many* n.

Proof. This follows from Corollary 3.3 by Kraft's inequality (1). □

THEOREM 3.6. *There is an unbounded search algorithm in which the cost of finding n is*

$$L(n) + \alpha(n) + \begin{cases} 2 & n \leq 1, \\ 1 & n > 1. \end{cases}$$

Proof. The algorithm, given in pseudo-Pascal in Algorithm 3, works in three stages. First, it tests $F(A(1))$, $F(A(2))$, $F(A(3))$, $\cdots$ until it finds a Y, say at $F(A(level+1))$. Thus we know that

$$A(level) \leq \text{location of the last X} < A(level+1). \tag{3}$$

If $level = 0$, we know that

$$0 \leq \text{location of the last X} < A(1) = 2,$$

and testing $F(1)$ tells us whether the answer is 0 or 1, in total of two tests.

If $level > 0$, suppose $F(n)$ is the last X; the first stage has then used $\alpha(n)+1$ tests and $level = \alpha(n)$. The second stage determines the location of the last X relative to the function $A_{level} = A_{\alpha(n)}$. We have

$$A(level) = A_{level}(level),$$

so applying the monotone, nondecreasing function α_{level} to the three parts of inequality (3) yields

$$level \leq \alpha(\text{location of the last X}) \leq \alpha_{level}(A(level+1)-1). \tag{4}$$

(Notice that we have changed the rightmost inequality to "$\leq$" by taking $A(level+1)-1$ in place of $A(level+1)$ in (3).) The second stage then tests $F(A_{level}(level+1))$, $F(A_{level}(level+2))$, $\cdots$ until a Y is found; this determines the location of the last X relative to A_{level}, at a cost of

$$\alpha_{level}(n) - level + 1 = \alpha_{\alpha(n)}(n) - \alpha(n) + 1$$

tests. However, we know from inequality (4) that we need not test any value beyond

$$\alpha_{level}(A(level+1)-1);$$

thus if tests up to and including this value fail to yield a Y, we need not make the next test—we already know that the answer will be Y. The number of tests in the second stage is thus

$$\begin{array}{ll} 0 & \text{if } n \leq 1, \\ \alpha_{\alpha(n)}(n) - \alpha(n) & \text{if } \alpha_{\alpha(n)}(n) = \alpha_{\alpha(n)}(A(\alpha(n)+1)-1), \\ \alpha_{\alpha(n)}(n) - \alpha(n) + 1 & \text{otherwise.} \end{array}$$

The third stage consists of applying the function *LevelSearch* from Algorithm 1, since we know *level* and we know the location of the last X relative to A_{level}. The number of tests used in this third stage is thus

$$\begin{array}{ll} 0 & n \leq 1, \\ L_{\alpha(n)}(n) & n > 1. \end{array}$$

ALGORITHM 3

Unbounded searching using the diagonal construction, in pseudo-Pascal. This method of searching is within $\alpha(n) + 2$ of the lower bound.

```
function DiagonalSearch(
    F: function(integer): [X, Y] { if F(n0) = Y then F(n) = Y for all n ≥ n0 }
    ): integer; { the location of the last X of F }

var
    level: integer; { level of the search }
    k: integer; { location of the last X relative to A_level;
                  that is, A_level(k) ≤ location < A_level(k + 1) }
    found: boolean; { flag to terminate the while loop }

function LevelSearch
⋮ { as in Algorithm 1 }
end; { LevelSearch }

begin { DiagonalSearch }

    level := 0;
    while F(A(level + 1)) = X do
        level := level + 1;

    { level = α(n); if level > 0 then
            A_level(level) = A(level) ≤ location of the last X < A(level + 1);
      that is,
            level ≤ α_level(location of the last X) ≤ α_level(A(level + 1) − 1) }

    if level = 0 then { the location of the last X is either 0 or 1 }
        if F(1) = X then return(1) else return(0)
    else begin { find the location of the last X relative to level level }
        found := false;
        k := level;
        while (k < α_level(A(level + 1) − 1)) and (not found) do
            if F(A_level(k + 1)) = X then
                k := k + 1
            else { terminate the loop }
                found := true;
        return(LevelSearch(level, k, f(x) = x))
        end

end; { DiagonalSearch }
```

TABLE 4
Some of the integer representations in the encoding C.

n	$C(n)$	n	$C(n)$	n	$C(n)$
0	00	8	11001000	64	1101010000000
1	01	9	11001001	127	1101010111111
2	100	10	11001010	128	11010110000000
3	101	15	11001111	255	11010111111111
4	1100000	16	11010000000	256	1101100000000000
5	1100001	31	11010001111	511	1101100011111111
6	1100010	32	110100100000	512	11011001000000000
7	1100011	63	110100111111	1000	11011001111101000

The total number of tests used is thus as stated in the theorem. □

Corollary 3.5 tells us that the diagonal search procedure in Algorithm 3 is within an additive factor $\alpha(n) + 2$ of being optimal. This is a great improvement over the level i search for all $i \geq 1$ because $\alpha(n)$ is *much* more slowly growing than $\alpha_i(n)$ as $n \to \infty$.

THEOREM 3.7. *There is a uniquely decipherable, irredundant, lexicographic, prefix-free code for the integers in which the number of bits used to represent n is*

$$L(n) + \alpha(n) + \begin{cases} 2 & n \leq 1, \\ 1 & n > 1. \end{cases}$$

Proof. Here is the code C corresponding to the diagonal search strategy:

$$C(n) = \begin{cases} 00 & \text{if } n = 0, \\ 01 & \text{if } n = 1, \\ 1^{\alpha(n)}01^{\alpha_{\alpha(n)}(n)-\alpha(n)}B_{\alpha(n)}(n) & \text{if } \alpha_{\alpha(n)}(n) = \alpha_{\alpha(n)}(A(\alpha(n)+1)-1), \\ 1^{\alpha(n)}01^{\alpha_{\alpha(n)}(n)-\alpha(n)}0B_{\alpha(n)}(n) & \text{otherwise.} \end{cases}$$

Some examples of the encoding C are shown in Table 4.

The unique decipherability of C follows from the correctness of the decoding algorithm, given in Algorithm 4, which decodes the prefix value and then applies the function *DecodeB* from Algorithm 2. The correctness of Algorithm 4 follows from the correctness of *DecodeB*, together with the evident correctness of the decoding of the prefix value. The lexicographic and prefix-free properties of C are also clear.

The code C is irredundant because $\sum_{n=0}^{\infty} 2^{-\operatorname{length}(C(n))} = 1$:

$$\begin{aligned} \sum_{n=0}^{\infty} 2^{-\operatorname{length}(C(n))} &= \frac{1}{4} + \frac{1}{4} + \sum_{n=2}^{\infty} 2^{-L(n)-\alpha(n)-1} \\ &= \frac{1}{2} + \frac{1}{2}\sum_{n=2}^{\infty} 2^{-L(n)-\alpha(n)} \\ &= \frac{1}{2} + \frac{1}{2}\sum_{m=1}^{\infty} \sum_{n\geq 1, \alpha(n)=m} 2^{-L(n)-m} \\ &= \frac{1}{2} + \frac{1}{2}\sum_{m=1}^{\infty} 2^{-m} \sum_{n\geq 1, \alpha(n)=m} 2^{-L(n)} \\ &= \frac{1}{2} + \frac{1}{2}\sum_{m=1}^{\infty} 2^{-m}. \end{aligned}$$

ALGORITHM 4

Decoding the codes in the diagonal construction, in pseudo-Pascal.

```
function DecodeDiagonal(
    b: BitString { the codeword to be decoded }
    ): integer; { the decoded value }

var
    level: integer; { level of the code }
    k: integer; { index of last bit of b processed }
    p: integer; { value of prefix part of b }
    found: boolean; { flag to terminate the while loop }

function DecodeB
:
: { as in Algorithm 2 }
end; { DecodeB }

begin { DecodeDiagonal }

    k := 0;
    while b_{k+1} = 1 do
        k := k+1;

    {k = α(n); b_1 = ··· = b_k = 1 and b_{k+1} = 0}

    if k = 0 then
        if b_2 = 1 then return(1) else return(0)
    else begin { compute the prefix for level level }
        found := false;
        level := k;
        p := k;
        k := k + 1; { skip the 0 }
        while (p < α_level(A(level + 1) − 1)) and (not found) do
            if b_{k+1} = 1 then begin
                k := k + 1;
                p := p + 1
                end
            else { terminate the loop }
                found := true;
        if found then
            k := k + 1; { skip the second 0 }
        return(DecodeB(level, p))
        end

end; { DecodeDiagonal }
```

by Theorem 3.1, and hence

$$= \frac{1}{2} + \frac{1}{2} = 1. \qquad \square$$

4. Extension to asymmetric costs. We have thus far assumed that the cost of a Y answer (or a zero-bit) is the same as the cost of an X answer (respectively, a one-bit). Suppose, however, that the cost of an X answer is a constant x and the cost of a Y answer is a constant y. This asymmetric form of unbounded searching is mentioned in [6] and investigated in [11].

If $x = 0$, the X answers are free and the optimal search strategy is to test $F(1)$, $F(2)$, $F(3)$, and so on, until we find the first Y; this costs exactly y, no matter where the first Y occurs. The optimal prefix-free, binary code, in this case, is to represent n by $1^{n-1}0$.

If $y = 0$, the Y answers are free and we can describe the level i search as testing $F(A_i(1))$, $F(A_i(2))$, $F(A_i(3))$, and so on, until we find a Y, say at $F(A_i(k+1))$. Then, we scan *backwards* linearly from $F(A_i(k+1))$ until we find the last X. This costs $x\alpha_i(n) + 1$, if the last X is at n, a cost that is clearly no further than $x\alpha_i(n)$ from optimal.

The only interesting case is when x and y are both nonzero. Kapoor and Reingold [11] developed an algorithm that is within $\Theta(\lg^* n)$ of a lower bound that they obtain in a manner parallel to that given for the symmetric case by Bentley and Yao [6]. We shall do much better here, proving the surprising result that our procedure *LevelSearch* is close to optimal in the asymmetric case, as well as the symmetric case. The analysis uses a generalization [11], [13] of Kraft's inequality (1): if $c(n)$ is the cost of an algorithm (encoding) when an X answer (respectively, one-bit) costs $x > 0$ and a Y answer (respectively, zero-bit) costs $y > 0$, then

$$\sum_{n=0}^{\infty} r^{-c(n)} \leq 1, \tag{5}$$

where r is the unique real number in the range $1 < r \leq 2$ satisfying $r^{-x} + r^{-y} = 1$.

We note, in passing, that unlike the symmetric case, there is no known partial converse to Kraft's inequality for the asymmetric case. As described in the introduction, the partial converse to Kraft's inequality for the symmetric case guarantees that when $c(1)$, $c(2)$, $c(3)$, $\cdots$ is a nondecreasing sequence of positive integers such that $\sum_{n=1}^{\infty} 2^{-c(n)} = 1$, there is a corresponding unbounded search algorithm of cost $c(n)$. In the asymmetric case the same conditions are insufficient to guarantee an algorithm: consider $x = 1$, $y = 2$; then $r = \phi = (1+\sqrt{5})/2$. The sequence 2, 4, 4, 5, 5, 6, 7, 8, 9, 10, 11, $\cdots$ is nondecreasing and

$$\phi^{-2} + \phi^{-4} + \phi^{-4} + \phi^{-5} + \phi^{-5} + \phi^{-6} + \phi^{-7} + \cdots = 1$$

(since $\phi^{-4} + \phi^{-5} = \phi^{-3}$ and $\phi^{-2} + \phi^{-3} + \phi^{-4} + \phi^{-5} + \phi^{-6} + \phi^{-7} + \cdots = 1$), but inspection of the possible roots of a search tree shows that there can be no asymmetric, lexicographic search tree in which $c(1) = 2$, $c(2) = 4$, $c(3) = 4$, $c(4) = 5$, $c(5) = 5$, $c(5) = 6$, and so on. Thus there can be no unbounded, asymmetric search algorithm with this cost function.

Let $x > 0$ and $y > 0$; define the *weighted length* of $B_i(n)$ to be

$$W_i(n) = x \times [\text{number of one-bits in } B_i(n)] + y \times [\text{number of zero-bits in } B_i(n)].$$

We thus have the following corollary to Theorem 2.6.

COROLLARY 4.1. *For all integers $i \geq 1$, there is a uniquely decipherable, irredundant, lexicographic, prefix-free, binary code for the nonnegative integers in which the weighted cost of the representation of n is*

$$W_i(n) + x\alpha_i(n) + \begin{cases} 2y & n = 0, \\ x + y & n = 1, \\ y & n > 1. \end{cases}$$

Proof. The expression given is precisely the weighted cost of C_i. □

Similarly, we thus have the following corollary to Theorem 2.5.

COROLLARY 4.2. *For all integers $i \geq 1$, there is an unbounded search algorithm in which the weighted cost of finding n is*

$$W_i(n) + x\alpha_i(n) + \begin{cases} 2y & n = 0, \\ x + y & n = 1, \\ y & n > 1. \end{cases}$$

Proof. This is the weighted cost of *search* (Algorithm 1) because, for $n > 1$, finding the location of the last X relative to A_i has weighted cost $x\alpha_i(n) + y$, after which the call to *LevelSearch* costs exactly the weighted cost of $B_i(n)$. For $n \leq 1$, tests are made at 2 and 1; for $n = 0$ these are both Y while for $n = 1$, the first is a Y and the second is an X. □

THEOREM 4.3. *For all integers $i \geq 1$ and all integers $m \geq 0$,*

$$\sum_{n \geq 1, \alpha_i(n) = m} r^{-W_i(n)} = 1.$$

Proof. The proof is by induction on i. For the basis we need to consider the case $m = 0$ and the case $i = 0$, $m > 0$. When $m = 0$, $\alpha_i(n) = 0$ only for $n \leq 1$; $B_i(1)$ is empty so $W_i(1) = 0$ and the sum is correct. When $i = 0$, we have $\alpha_0(n) = m$ for $n = 2m$ or $n = 2m + 1$; $B_0(2m) = 0$ and $B_0(2m+1) = 1$ giving $W_0(2m) = y$ and $W_0(2m+1) = x$, so the sum is simply $r^{-x} + r^{-y} = 1$ by the definition of r.

Suppose the theorem holds for $i - 1$ and $m > 0$. In this case $n > 1$, so

$$B_i(n) = B_i(\alpha_{i-1}(n))B_{i-1}(n)$$

making

$$W_i(n) = W_i(\alpha_{i-1}(n)) + W_{i-1}(n).$$

Following the proof of the lemma, observe that by the definition of $\alpha_i(n)$ we have $\alpha_i(\alpha_{i-1}(n)) = \alpha_i(n) - 1$, so that by the definition of $W_i(n)$

$$\begin{aligned} \sum_{n \geq 1, \alpha_i(n) = m} r^{-W_i(n)} &= \sum_{n \geq 1, \alpha_i(\alpha_{i-1}(n)) = m-1} r^{-W_{i-1}(n) - W_i(\alpha_{i-1}(n))} \\ &= \sum_{k \geq 1, \alpha_i(k) = m-1} r^{-W_i(k)} \sum_{n \geq 1, \alpha_{i-1}(n) = k} r^{-W_{i-1}(n)}, \end{aligned}$$

by substituting k for $\alpha_{i-1}(n)$ and rewriting. But, by induction, the inner sum is 1, giving

$$\begin{aligned} &= \sum_{k \geq 1, \alpha_i(k) = m-1} r^{-W_i(k)} \\ &= 1, \end{aligned}$$

again by induction. □

Thus $\sum_{n=1}^{\infty} r^{-W_i(n)}$ diverges and we have the following corollary.

COROLLARY 4.4. *For all integers $i \geq 1$, $W_i(n)$ is a lower bound on the asymmetric-cost version of the unbounded searching problem of Bentley and Yao* [6] *and on the asymmetric-cost version of prefix-free, binary encodings of the nonnegative integers* [7] *(see* [14]*), in the sense that if an unbounded search algorithm uses cost $c(n)$ to locate n (or, equivalently, a prefix-free code uses cost $c(n)$ to encode n), then $c(n) > W_i(n)$ for infinitely many n.*

Hence for any *level* ≥ 1, the search of Algorithm 1 is within an additive factor $x\alpha_{level}(n) + \max\{2y, x+y\}$ of being optimal for the asymmetric case. A similar statement holds for the diagonal search of Algorithm 3 because Corollaries 4.1 and 4.2, Theorem 4.3, and Corollary 4.4 all hold for the diagonal case as well. We do not dwell on this here as these results are subsumed by those of [20].

5. What is an optimal algorithm for unbounded searching? Our intuition tells us that it ought to be possible to capitalize on the asymmetry, yet the same search algorithm/encoding appears to be equally close to optimal for both the symmetric case and *any* asymmetric case. The difficulty is in the meaning of the phrase "close to optimal." Bentley and Brown [5] call it "very delicate," but "peculiar" might be a better description.

To understand why it is peculiar, we return to the symmetric case and observe that it is really quite simple to construct a naive algorithm/encoding that is within $\alpha_i(n) + 1$ of a lower bound:[2] Test $F(A_i(1))$, $F(A_i(2))$, $F(A_i(3))$, $\cdots$ until we get a Y answer at $A_i(\alpha_i(n)+1)$, where the location of the last X is n. Then, do a binary search on the range $A_i(\alpha_i(n))+1, \cdots, A_i(\alpha_i(n)+1)-1$. Let the cost of this binary search be $S_i(n)$, $S_i(n) = \lceil \lg(A_i(\alpha_i(n)+1) - A_i(\alpha_i(n))) \rceil$, so the entire algorithm thus described requires $S_i(n) + \alpha_i(n) + 1$ tests for $n > 1$. By the nature of binary trees,

$$\sum_{\text{all leaves}} 2^{-\text{distance to the leaf}} = 1,$$

so that

$$\sum_{A_i(\alpha_i(n)) < k < A_i(\alpha_i(n)+1)} 2^{-S_i(k)} = 1,$$

and hence

$$\sum_{n=0}^{\infty} 2^{-S_i(n)} = \infty,$$

making $S_i(n)$ a lower bound. This algorithm is thus "within $\alpha_i(n)+1$ of optimal," but the algorithm is absurd! $S_i(n)$ is ridiculously large for values of n just above $A_i(\alpha_i(n))$. It is so large that for $i > 2$, the cost of the search when the location of the last X is at $n = A_i(\alpha_i(n)) + 1$ cannot be bounded by any polynomial function of n; for the diagonal version, it cannot be bounded by any primitive recursive function of n.[3]

[2] This construction, which is found in [19], works for *any* unbounded, computable function $g(n)$ and its "inverse" (in the same sense that α_i and A_i are inverses) $G(n) = g^{-1}(n)$; also, see [3, Thm. 34].

[3] The difficulty here is the use of binary search. Even though binary search is the optimal search strategy on a known interval, the unbounded search problem requires measuring the cost of the search as a function of the *location in which the answer is found.* In this case, binary search is decidedly suboptimal over a wide range of answers.

Comparing the cost of our level i algorithm with the lower bound $S_i(n)$, we discover that

$$\limsup_{n\to\infty} \frac{L_i(n) + \alpha_i(n) + 1}{S_i(n)} = 1,$$

while comparing the cost of the naive search to the lower bound $L_i(n)$ gives

$$\limsup_{n\to\infty} \frac{S_i(n) + \alpha_i(n) + 1}{L_i(n)} = \infty.$$

That is, our algorithms are near-optimal *vis-à-vis* the absurd algorithm's lower bound, but the naive search is not near-optimal *vis-à-vis* our lower bounds. We will see that our algorithms are near-optimal compared to *any* nontrivial lower bound, for $i \geq 2$.

Since

$$\sum_{n=1}^{\infty} 2^{-\lg n} = \infty,$$

any search algorithm must use at least $\lg n$ tests, in the sense that if $c(n)$ is the cost of a search algorithm/encoding, we must have

$$\limsup_{n\to\infty} \frac{c(n)}{\lg n} \geq 1,$$

for otherwise we could not have $c(n) > \lg n$ infinitely often. Lower bounds "below" $\lg n$ are thus of no interest; for example, $b(n) = 1$ is a lower bound because $\frac{1}{2} + \frac{1}{2} + \cdots = \infty$, but it is a trivial lower bound. This situation suggests the following *ad hoc* definition.

DEFINITION. Given that $\sum_{n=0}^{\infty} 2^{-b(n)}$ diverges, call $b(n)$ a *nontrivial lower bound* if, in addition,

$$\limsup_{n\to\infty} \frac{\lg n}{b(n)} \leq 1.$$

Notice that $S_i(n)$ and $L_i(n)$ are nontrivial lower bounds according to this definition, but $b(n) = 1$ is not. This definition of near-optimality is *ad hoc* in that we could use, say, $\lg n + O(\log \log n)$ instead of $\lg n$. Because our definition is intended to forbid absurd behavior, we chose the simplest function that would have that effect; more complex choices could be made, but $\lg n$ is the cleanest.

DEFINITION. If an algorithm/encoding has cost $c(n)$, we say that the algorithm/encoding is *within* $f(n)$ *of being optimal* if $c(n) - f(n)$ is a lower bound *and* $\limsup_{n\to\infty} c(n)/b(n) = 1$ for *any* nontrivial lower bound $b(n)$.

It is easy to show by induction that $L_i(n) = \lg n + O(\lg \lg n)$ for all $i \geq 1$, so

$$\lim_{n\to\infty} \frac{L_i(n) + \alpha_i(n) + 1}{\lg n} = 1$$

for $i \geq 2$. (This fails for $i = 1$ because $L_1(n) = \alpha_1(n) = \lfloor \lg n \rfloor$, making the limit 2, not 1.) If $b(n)$ is a nontrivial lower bound,

$$\limsup_{n\to\infty} \frac{\lg n}{b(n)} \leq 1;$$

multiplying this inequality by the previous equation gives

$$\limsup_{n\to\infty} \frac{L_i(n) + \alpha_i(n) + 1}{b(n)} \le 1.$$

However, $L_i(n) + \alpha_i(n) + 1$ is the cost of an algorithm/encoding, and $b(n)$ is a lower bound; thus

$$\limsup_{n\to\infty} \frac{L_i(n) + \alpha_i(n) + 1}{b(n)} \ge 1.$$

Hence,

$$\limsup_{n\to\infty} \frac{L_i(n) + \alpha_i(n) + 1}{b(n)} = 1$$

for *any* nontrivial lower bound. We are thus justified in claiming that our level i algorithm is within $\alpha_i(n) + 1$ of being optimal, for $i \ge 2$; similarly, our diagonal algorithm is optimal to within $\alpha(n) + 2$ (this is a special case of [20, Cor. 3 and Thm. 4]). We are using the term "optimal to within $f(n)$" in a much stronger sense than in [3] or [6], although their algorithms (except the algorithms implicit in [3, Thm. 34]) are near-optimal in this strong sense. In fact, any search algorithm that is near-optimal in the sense of [6] and for which $c(n) = \lg n + o(\lg n)$ is optimal in the strong sense.[4]

Finally, suppose we are given two unbounded search algorithms, each of which is known to be within $f(n)$ of being optimal in the strong sense defined above. How can we determine which algorithm, if either, is superior? For example, Bentley and Yao's ultimate algorithm (or, equivalently, Knuth's encoding R or our level $i = 2$ algorithm/encoding) and Beigel's nonconstructive algorithm (see [3, Thms. 23 and 32]) are each within $\Theta(\lg^* n)$ of being optimal in the strong sense, but which is better? They are difficult to compare directly because of the different forms of the cost functions—especially the presence/absence of floor and ceiling functions.

In the same way we look to the slowness of divergence to provide better lower bounds, so we may look to the slowness of convergence to indicate the better algorithm. Specifically, if $\sum_{n=0}^{\infty} 2^{-b(n)}$ diverges, the more slowly it diverges the larger (and hence better) the lower bound $b(n)$. Similarly, if $\sum_{n=0}^{\infty} 2^{-c(n)}$ converges to 1, the more slowly it converges, the smaller (and hence better) the cost function $c(n)$. We therefore propose to compare various algorithms based on the speed of convergence of

[4] Beigel [4] has pointed out that the construction of "absurd" algorithms [3], [19] that are near-optimal in the sense of [6] can be applied, with modification, to our stronger notion of near-optimality. The idea is this: let $\mathcal{U}$ be an unbounded search algorithm that makes at most $\lg n + 2\lg\lg n$ tests; for example, Bentley and Yao's ultimate search or our level 2 algorithm could be chosen as $\mathcal{U}$. Now, let $g(n)$ be any nondecreasing, computable function, $g(n) = o(\log\log n)$, and let $G(n) = g^{-1}(n)$ be the inverse of $g(n)$ in the same sense that $A_i(n) = \alpha_i^{-1}(n)$. Consider the algorithm $\mathcal{V}$ constructed from $\mathcal{U}$ as follows. $\mathcal{V}$ tests $F(G(1)), F(G(2)), \cdots$ until it gets a Y answer at $F(G(g(n)+1))$. Then, $\mathcal{V}$ uses algorithm $\mathcal{U}$ to search the interval $[G(g(n)), G(g(n)+1))$, but avoiding any redundant tests. The algorithm $\mathcal{V}$ is within $g(n)$ of being optimal by the same argument that proves the absurd algorithm is near-optimal in the Bentley–Yao sense; however, because $\mathcal{V}$ uses no more than $\lg n + 2\lg\lg n + g(n) + 1$ tests it is also optimal in our strong sense as well. This construction has the twin disadvantages of requiring the application of an unbounded search algorithm to a finite search problem and needing to avoid redundant tests. Since the construction leaves no way to determine the number of tests used as a function of n, it does not render our stronger notion of near-optimality peculiar because the algorithm $\mathcal{V}$ cannot be regarded as absurd.

$\sum_{n=0}^{\infty} 2^{-c(n)}$. Given two algorithms with cost functions $c_1(n)$ and $c_2(n)$, respectively, we compute the tail of the series $\sum_{n=0}^{\infty} 2^{-c_1(n)}$

$$T_1(x) = \sum_{n \geq x} 2^{-c_1(n)}$$

and compare it to the tail of the series $\sum_{n=0}^{\infty} 2^{-c_2(n)}$

$$T_2(x) = \sum_{n \geq x} 2^{-c_2(n)}$$

to see which goes to zero more slowly as $x \to \infty$. This method of comparison does not yield a total ordering on unbounded search algorithms. Rather, it magnifies differences between algorithms of nearly identical behavior, elucidating subtle differences.

First, we compute the tail of the series for our level i algorithm; when $i = 2$ this is essentially the same as Knuth's encoding R and Bentley and Yao's ultimate algorithm.

$$\begin{aligned} T_{A_i}(A_i(n_0)) &= \sum_{n \geq A_i(n_0)} 2^{-L_i(n)-\alpha_i(n)-1} \\ &= \sum_{i=n_0}^{\infty} \sum_{\alpha_i(j)=i} 2^{-L_i(j)-i-1} \\ &= \sum_{i=n_0}^{\infty} 2^{-i-1} \sum_{\alpha_i(j)=i} 2^{-L_i(j)} \\ &= \sum_{i=n_0}^{\infty} 2^{-i-1} \end{aligned}$$

by Theorem 4.3, and so

$$= 2^{-n_0}.$$

Hence,

$$T_{A_i}(x) \approx 2^{-\alpha_i(x)}.$$

Recall that $\alpha_2(x) = \lg^* x$ so that our level 2 search algorithm, Knuth's encoding R, and Bentley and Yao's ultimate algorithm all have a tail

$$T_{A_2}(x) \approx 2^{-\lg^* x}.$$

Now, we compute the tail for Beigel's algorithm [3, Thm. 23] which has cost

$$C_B(n) = \left\lceil \left(\sum_{i=1}^{\lg^* n} \lg^{(i)} n \right) - (\lg \lg(e - \delta)) \lg^* n \right\rceil,$$

for any δ, $0 < \delta < e$, where e is the base of the natural logarithms. For convenience, let

$$q = \frac{1}{\ln(e - \delta)} > 1.$$

We have

$$T_B(tower(n_0)) \approx \frac{q^{-n_0}}{q-1}$$

giving

$$T_B(x) \approx \frac{q^{-\lg^* x}}{q-1}.$$

Comparing $T_{A_2}(x)$ with $T_B(x)$, we find that, as $x \to \infty$, $T_B(x) \to 0$ more slowly than $T_{A_2}(x) \to 0$ when $q < 2$, that is, when $\delta < e - \sqrt{e} \approx 1.06956$. Of course, for $i > 2$, $T_{A_i}(x) \to 0$ *much* more slowly than $T_B(x) \to 0$, as $x \to \infty$, for any choice of δ.

6. The asymmetric case revisited. Since $\sum_{n=1}^{\infty} r^{-\log_r n}$ diverges, $\log_r n$ is a lower bound for the asymmetric case by the generalized Kraft's inequality (5). Thus we are not interested in lower bounds below $\log_r n$, just as in the symmetric case we are not interested in lower bounds below $\lg n$.

Our level i algorithm/encoding, as presented so far, is within about $x\alpha_i(n)$ of being optimal in the original sense of [6], but it is *not* that close to being optimal in the stronger sense defined in the previous section. Specifically,

$$\limsup_{n\to\infty} \frac{W_i(n)}{\log_r n} = \max\{x, y\} \lg r$$

by the following argument. Clearly,

$$\begin{aligned} W_i(n) &\leq \max\{x, y\} L_i(n) \\ &= \max\{x, y\} \lg n + O(\lg \lg n) \\ &= (\max\{x, y\} \lg r) \log_r n + O(\lg \lg n), \end{aligned}$$

so $\max\{x, y\} \lg r$ is an upper bound on the lim sup. However, considering the two subsequences $n = 2^k - 1$ and $n = 2^k$, for which the component $B_1(n)$ of $C_i(n)$ contains about $\lg n$ ones or $\lg n$ zeros, respectively, we find that

$$\limsup_{n\to\infty} \frac{W_i(n)}{\log_r n} \geq \max\{x, y\} \lg r.$$

Thus $\limsup_{n\to\infty} W_i(n)/\log_r n$ is finite for any choice of x and y, but it can be arbitrarily large. In other words, our level i algorithm/encoding is within a constant *multiplicative* factor of being optimal in the asymmetric case, not within a slowly growing *additive* factor.

To obtain such a near-optimal algorithm/encoding for the asymmetric case, we must redefine $A_0(n)$ and $\alpha_0(n)$ as

$$A_0(n) = \lceil r^y n \rceil,$$

$$\alpha_0(n) = \lfloor n/r^y \rfloor;$$

the recursive definitions of $A_i(n)$ and $\alpha_i(n)$ for $i > 0$ are unchanged. The algorithms remain identical to those in the symmetric case, except at the two boundaries where the recursion bottoms out. (Since the *functions* $A_i(n)$ have changed, the probes

are at entirely different places and the behavior of the algorithms is correspondingly different; only the *form* of the algorithms is identical because the ideas are similar.) One of these boundaries is at level 0, when

$$A_0(u) \leq \text{location of the last X} < A_0(u+1),$$

and we use the optimal bounded lopsided search algorithm of [11, §2.2] to determine the exact location of the last X in the range $[A_0(u), A_0(u+1)-1]$. The cost of this algorithm is approximately

$$\log_r (A_0(u+1) - A_0(u)) = \log_r (\lceil r^y(u+1)\rceil - \lceil r^y u\rceil) \approx y.$$

The other boundary is at $u = 0$, when

$$A_i(0) \leq \text{location of the last X} < A_i(1).$$

We have $A_i(0) = 1$ and $A_i(1) = \lceil r^y \rceil$ for all $i \geq 0$, so again we use the optimal bounded search algorithm of [11] to determine the exact location of the last X; the cost is approximately

$$\log_r \lceil r^y \rceil \approx y.$$

We define

$$L_0(0) = 0,$$

and for $n \geq 1$,

$$L_0(n) = \begin{cases} \text{the exact cost to search for } n \text{ in the interval} \\ [A_0(\alpha_0(n)), A_0(\alpha_0(n)+1)-1] \text{ using the optimum bounded lopsided search algorithm} \\ \text{of [11, §2.2].} \end{cases}$$

By the nature of asymmetrically weighted binary trees,

$$\sum_{A_0(m) \leq n < A_0(m+1)} r^{-L_0(n)} = 1,$$

or, equivalently,

$$\sum_{\alpha_0(n)=m} r^{-L_0(n)} = 1.$$

We also define

$$L_i(n) = \begin{cases} 0 & n = 0, \\ L_0(n) & 1 \leq n < \lceil r^y \rceil, \\ L_{i-1}(n) + L_i(\alpha_{i-1}(n)) & n \geq \lceil r^y \rceil, \end{cases}$$

and

$$L(n) = \begin{cases} 0 & n = 0, \\ L_0(n) & 1 \leq n < \lceil r^y \rceil, \\ L_{\alpha_i(n)}(n) + x(\alpha_{\alpha_i(n)}(n) - \alpha_i(n)) & n \geq \lceil r^y \rceil \text{ and } \alpha_{\alpha_i(n)}(n) = \\ & \qquad \alpha_{\alpha_i(n)}(A_i(\alpha_i(n)+1)-1), \\ L_{\alpha_i(n)}(n) + x(\alpha_{\alpha_i(n)}(n) - \alpha_i(n)) + y & \text{otherwise.} \end{cases}$$

An induction shows that for $i \geq 0$,

$$\sum_{\alpha_i(n)=m} r^{-L_i(n)} = 1,$$

making the $L_i(n)$ lower bounds on the asymmetric unbounded search problem; similarly, $L(n)$ is a lower bound. As before, we have $L_i(n) = \log_r n + O(\log\log n)$ and $L(n) = \log_r n + O(\log\log n)$ so that these lower bounds are nontrivial. The corresponding algorithms/encodings are thus within an additive factor $x\alpha_i(n) + \max\{2y, x+y\}$ of being optimal in the strong sense of the previous section. When $i = 2$, this algorithm/encoding and the corresponding lower bound are almost identical to the asymmetric unbounded search algorithm and lower bound of [11].

Note that we could define

$$A_0(n) = \lceil r^{ty} n \rceil,$$

$$\alpha_0(n) = \lfloor n / r^{ty} \rfloor,$$

for any number $t \geq 1$, and a similar optimality result holds. Thus, for example, if $x > y$ we could take $t = x/y$. The larger the value of t, the closer the leading terms in $L_i(n)$ and $L(n)$ are to $\log_r n$, for $i \geq 1$.

Acknowledgments. We wish to express our gratitude to Richard Beigel who provided numerous insightful comments about our results; many improvements to this paper resulted from his suggestions. We also thank Herbert Edelsbrunner, Micha Sharir, and Kenneth J. Supowit for their helpful remarks.

REFERENCES

[1] W. Ackermann, *Zum Hilbertschen aufbau der reellen Zahlen*, Math. Ann., 99 (1928), pp. 118–133.

[2] R. P. Agnew, *A slowly divergent series*, Amer. Math. Monthly, 54 (1947), pp. 273–274.

[3] R. Beigel, *Unbounded searching algorithms*, SIAM J. Comput., 19 (1990), pp. 522–537.

[4] ———, Personal communication.

[5] J. L. Bentley and D. J. Brown, *A general class of resource tradeoffs*, J. Comput. Systems Sci., 5 (1982), pp. 214–238.

[6] J. L. Bentley and A. C.-C. Yao, *An almost optimal algorithm for unbounded searching*, Inform. Process. Lett., 5 (1976), pp. 82–87.

[7] P. Elias, *Universal codeword sets and representations of the integers*, IEEE Trans. Inform. Theory, 21 (1975), pp. 194–203.

[8] S. Even and M. Rodeh, *Economical encoding of commas between strings*, Comm. ACM, 21 (1978), pp. 315–317.

[9] A. S. Goldstein and E. M. Reingold, *A Fibonacci version of Kraft's inequality applied to discrete unimodal search*, Report Number UIUCDCS-R-90-1721, Department of Computer Science, University of Illinois, Urbana, IL, April, 1990.

[10] S. Hart and M. Sharir, *Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151–177.

[11] S. Kapoor and E. M. Reingold, *Optimum lopsided binary trees*, J. Assoc. Comput. Mach., 36 (1989), pp. 573–590.

[12] L. V. Kale and V. Saletore, *Parallel state-space search for a first solution with consistent linear speedups*, Report Number UIUCDCS-R-89-1549, Department of Computer Science, University of Illinois, Urbana, IL, 1989.

[13] R. M. Karp, *Minimum-redundancy coding for the discrete noiseless channel*, IRE Trans. Inform. Theory, 7 (1961), pp. 27–39.

[14] D. E. Knuth, *Supernatural Numbers*, in The Mathematical Gardner, D. A. Klarner, ed., Wadsworth International, Belmont, CA, 1981, pp. 310–325.

[15] L. G. Kraft, *A Device for Quantizing, Grouping and Coding Amplitude Modified Pulses*, Master's thesis, Electrical Engineering Department, Massachusetts Institute of Technology, Cambridge, MA, 1949.

[16] V. I. Lebenshtein, *On the redundancy and delay of decodable coding of natural numbers*, Systems Theory Research, 20 (1968), pp. 149–155.

[17] S. K. Leung-Yan-Cheong and T. M. Cover, *Some equivalences between Shannon entropy and Kolmogorov complexity*, IEEE Trans. Inform. Theory, 24 (1978), pp. 331–338.

[18] Z. Li and E. M. Reingold, *Solution of a divide-and-conquer maximin recurrence*, SIAM J. Comput., 18 (1989), pp. 1188–1200.

[19] J.-C. Raoult and J. Vuillemin, *Optimal unbounded search strategies,* Rapport de Recherche 33, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1979.

[20] E. M. Reingold and X. Shen, *More nearly optimal algorithms for unbounded searching, Part II: The transfinite case*, SIAM J. Comput., this issue , pp. 184–208.

[21] J. Rissanen, *A universal prior for integers and estimation by minimum description length*, Ann. Statist., 11 (1983), pp. 416–431.

[22] J. S. Rohl, *Recursion via Pascal*, Cambridge University Press, Cambridge, England, 1984.

[23] V. Saletore and L. V. Kale, *Obtaining first solutions fast in AND-OR parallel execution of logic programs*, in Proc. North American Conference on Logic Programming, Cleveland, OH, 1989, pp. 390–408.

[24] Q. F. Stout, *Improved prefix encodings of the natural numbers*, IEEE Trans. Inform. Theory, 26 (1980), pp. 607–609.

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

SIAM J. COMPUT.
Vol. 20, No. 1, pp. 184–208, February 1991
© 1991 Society for Industrial and Applied Mathematics
011

MORE NEARLY OPTIMAL ALGORITHMS FOR UNBOUNDED SEARCHING, PART II: THE TRANSFINITE CASE*

EDWARD M. REINGOLD† AND XIAOJUN SHEN‡

Abstract. Given a function $F : N^+ \to \{X, Y\}$ with the property that if $F(n_0) = Y$ then $F(n) = Y$ for all $n > n_0$, the *unbounded search problem* is to use tests of the form "is $F(i) = X$?" to determine the smallest n such that $F(n) = Y$; the "cost" of a search algorithm is a function $c(n)$, the number of such tests used when the location of the first Y is n. In Part I of this paper it is shown how to construct an infinite sequence of algorithms, each of which is much closer to optimality than its predecessor. Diagonalizing over this sequence yields a new algorithm that is far better than any of the algorithms in the sequence: this "omega-th" algorithm is within an additive factor of $\alpha(n) + 2$ of the corresponding lower bound, where $\alpha(n)$ is a functional inverse of Ackermann's function—an *extremely* slowly growing function. In this paper the construction techniques are generalized to get dramatically better algorithms and lower bounds *ad infinitum*. Specifically, for each ordinal $\iota \leq \epsilon_0$, an algorithm is given that is dramatically closer to optimality than the algorithm corresponding to a smaller ordinal. All algorithms constructed for $\iota < \epsilon_0$ are proved to be optimal in a strong sense. Parallel results for the asymmetric case are also given.

Key words. unbounded search, prefix-free codes, optimal algorithms, Ackermann's function, inverse Ackermann's function, Kraft's inequality, ordinal numbers, Grzegorczyk hierarchy

AMS(MOS) subject classifications. 68Q25, 68Q20, 94B45, 04A10, 26A12

1. Introduction. The *unbounded search problem*, introduced by Bentley and Yao [2], is to determine the location of the first Y value of a function F from the positive integers to the set {X, Y}, when F has the property that if $F(n_0) = Y$, then $F(n) = Y$ for all $n > n_0$. The "cost" of the search algorithm is a function $c(n)$, which specifies the number of tests "is $F(i) = X$?" used when the location of the first Y is n. Bentley and Yao point out the equivalence of this problem and table lookup in an infinite ordered table, and they also show its connection to the construction of prefix-free, binary encodings of the integers in which the codeword for n has $c(n)$ bits.

In Part I of this paper [15], we use Ackermann's functions to construct an infinite sequence of algorithms/encodings, and a corresponding sequence of lower bounds, in which the zeroth algorithm is a form of linear search, the first algorithm is a form of binary search, the second algorithm is (essentially) Bentley and Yao's ultimate algorithm, and the improvement from the ith algorithm to the $(i + 1)$st algorithm is equally dramatic for any i. Diagonalizing over the zeroth, first, second, and so on gives us the omega-th algorithm, which is within an additive factor of $\alpha(n) + 2$ of the corresponding lower bound; $\alpha(n)$ is the functional inverse of Ackermann's function, an *extremely* slowly growing function. In this paper we show how to continue the construction to get dramatically better algorithms/encodings and lower bounds *ad infinitum* beyond the omega-th. Specifically, for each ordinal number $\iota \leq \epsilon_0$, we give an algorithm and corresponding lower bound that are much closer together than any algorithm/lower bound pair corresponding to a smaller ordinal. When ι is a successor ordinal, the corresponding algorithm is based on the predecessor algorithm; when ι

* Received by the editors April 30, 1989; accepted for publication (in revised form) June 2, 1990.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ Computer Science and Telecommunications Program, University of Missouri-Kansas City, Kansas City, Missouri 64110. Part of this work was done at the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This author's research was partially supported by Office of Naval Research contract N00014-86K-0416 and by the East China Institute of Technology, Nanjing, China.

is a limit ordinal, the corresponding algorithm is based on a diagonalization over the fundamental sequence for ι. In §4 we show that for $\iota < \epsilon_0$ these algorithms are optimal in the strong sense of [15] by proving that the number of diagonalizations used for the ιth algorithm is bounded by a simple function obtained from the Cantor normal form of ι; this result and the techniques used to obtain it are of independent interest.

We also show how to obtain parallel results for the asymmetric case in which the cost of a Y answer and the cost of an X answer are not the same. This closes the gap between the cost of the best algorithm and the best lower bound to a razor-sharp edge for both the symmetric and asymmetric cases.

As in Part I, it is convenient for notational simplicity to insist that $F(0) = \mathrm{X}$ and to state the unbounded search problem as that of finding the greatest integer $n \geq 0$ such that $F(n) = \mathrm{X}$; in other words, instead of looking for the first Y, we look for the last X. If the last X occurs at position $n \geq 0$, the search for the last X corresponds to the encoding of the integer n.

2. Construction ad infinitum. Following our construction in Part I, we might use Ackermann's function $A(n)$ ([1], but see [15]) to define, say,

$$\hat{A}_i(n) = \begin{cases} A(n) & i = 0, \\ \hat{A}_{i-1}^{(n)}(1) & i \geq 1, \end{cases}$$

and the corresponding inverse functions. This would lead, with the level-by-level construction, to a better algorithm and a better lower bound for each i. Of course, we could then diagonalize over the $\hat{A}_i(n)$ to obtain $\hat{A}(n) = \hat{A}_n(n)$, the corresponding inverse function, the corresponding diagonal algorithm, and the corresponding lower bound. We can do this again and again, ultimately diagonalizing over the functions $A(n), \hat{A}(n), \cdots$. Then we could start over and continue *ad nauseum*!

To make this infinite sequence of level-by-level and diagonal constructions precise, we use the language of ordinal numbers (see, for example, [8]). For convenience of notation, we will write $\mathrm{pred}(\beta)$ to represent the predecessor ordinal of a successor ordinal β; that is, $\beta = \mathrm{pred}(\beta) + 1$. Similarly, if β is the limit ordinal for the sequence $\{\beta_k\}$, we will write $\mathrm{pred}(\beta, k)$ for β_k. It is important that we be consistent when choosing limit sequences for limit ordinals; we do not, for instance, want to use one limit sequence when defining the algorithm and a different sequence when constructing the corresponding lower bound. We will use the fixed fundamental sequences defined by Wainer [19] (see also [16]).

DEFINITION (Wainer [19]). A *fixed fundamental sequence* is defined for every limit ordinal $\sigma \leq \epsilon_0$ as $\mathrm{pred}(\sigma, 0), \mathrm{pred}(\sigma, 1), \mathrm{pred}(\sigma, 2), \cdots$ where

(i) If $\sigma = \omega^\eta \cdot \beta$, η and β are both successor ordinals, then $\mathrm{pred}(\sigma, n) = \omega^\eta \cdot \mathrm{pred}(\beta) + \omega^{\mathrm{pred}(\eta)} \cdot n$.

(ii) If $\sigma = \omega^\eta \cdot \beta$, $\eta < \sigma$ is a limit ordinal, and β is a successor ordinal, then $\mathrm{pred}(\sigma, n) = \omega^\eta \cdot \mathrm{pred}(\beta) + \omega^{\mathrm{pred}(\eta, n)}$.

(iii) If $\sigma = \epsilon_0$, then $\mathrm{pred}(\sigma, 0) = 1$ and $\mathrm{pred}(\sigma, n) = \omega^{\mathrm{pred}(\sigma, n-1)}$.

It is convenient to extend this definition slightly to include successor ordinals by defining $\mathrm{pred}(\sigma + 1, n) = \sigma$ and $\mathrm{pred}(0, n) = 0$, for all $n \geq 0$. Although fundamental sequences for much larger initial segments of the ordinals are also available (see [5] or [17]), in this paper we will not concern ourselves with ordinals beyond ϵ_0.

As a final comment about ordinal numbers, we note that there is an elegant, easy-to-implement tree representation of the ordinals that goes well beyond ϵ_0 [3]. Thus the use of ordinals as a data type in our algorithms does not unduly complicate their implementation.

DEFINITION. The *generalized Ackermann's function* $A_\iota(n)$ is defined for every ordinal $\iota \leq \epsilon_0$ and nonnegative integer n by

$$A_\iota(n) = \begin{cases} 2n & \text{if } \iota = 0, n \geq 1, \\ A^{(n)}_{\text{pred}(\iota)}(1) & \text{if } \iota \text{ is a successor ordinal,} \\ A_{\text{pred}(\iota,n)}(n) & \text{if } \iota \text{ is a limit ordinal.} \end{cases}$$

For convenience, we define $A_\iota(0) = 1$ for all ordinals $\iota \geq 0$. Similar functions arise in various other contexts; see [6], [10], [13], [16], [18], or [19].

For $\iota < \omega$, these functions are exactly the ordinary Ackermann's functions defined in §2 of Part I [15]; also $A_\omega(n) = A(n)$, the diagonal Ackermann's function defined in §3 of Part I. Notice that $A_\iota(1) = 2$ and $A_\iota(2) = 4$ for all ι. It is easy to show, by transfinite induction on ι, that A_ι is a strictly increasing function, and that $A_\iota(n)$ is majorized by $A_\beta(n)$ whenever $\iota < \beta$; see [10], [16], or [19], for instance.

DEFINITION. The *inverse generalized Ackermann's function* is defined for every ordinal $\iota \leq \epsilon_0$ and nonnegative integer n by

$$\alpha_\iota(n) = \begin{cases} \lfloor n/2 \rfloor & \text{if } \iota = 0, \\ \text{least } j \text{ such that } \alpha^{(j)}_{\text{pred}(\iota)}(n) \leq 1 & \text{if } \iota \text{ is a successor ordinal,} \\ \text{greatest } j \text{ such that } \alpha_{\text{pred}(\iota,j)}(n) \geq j & \text{if } \iota \text{ is a limit ordinal.} \end{cases}$$

Equivalently, for a limit ordinal ι, $\alpha_\iota(n)$ could be defined as the greatest j such that $A_{\text{pred}(\iota,j)}(j) \leq n$. As expected,

$$\alpha_\iota(n) = A_\iota^{-1}(n) = \text{greatest } x \geq 1 \text{ such that } A_\iota(x) \leq n,$$

$$\alpha_\iota(A_\iota(n)) = n,$$

and

$$A_\iota(\alpha_\iota(n)) = \text{least } x \geq 1 \text{ such that } \alpha_\iota(x) = \alpha_\iota(n).$$

Of course, for $\iota < \omega$, these functions are exactly the inverse Ackermann's functions defined in §2 of Part I and $\alpha_\omega(n) = \alpha(n)$, the diagonal inverse Ackermann's function defined in §3 of Part I.

DEFINITION. For any ordinal $\iota \leq \epsilon_0$, we define the *generalized length function* $L_\iota(n)$ for nonnegative integers n by

$$L_0(n) = \begin{cases} 0 & n \leq 1, \\ 1 & n > 1, \end{cases}$$

$$L_\iota(n) = \begin{cases} 0 & n \leq 1, \\ L_{\text{pred}(\iota)}(n) + L_\iota(\alpha_{\text{pred}(\iota)}(n)) & n > 1, \end{cases}$$

for successor ordinals ι, and

$$L_\iota(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ L_{\text{pred}(\iota,\alpha_\iota(n))}(n) & \text{if } \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) = \\ \quad +\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n) & \quad \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(A_\iota(\alpha_\iota(n)+1)-1), \\ L_{\text{pred}(\iota,\alpha_\iota(n))}(n) & \\ \quad +\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n) + 1 & \text{otherwise,} \end{cases}$$

for limit ordinals ι.

Again, for $\iota < \omega$, these are exactly the functions defined in §2 of Part I, and $L_\omega(n)$ is exactly $L(n)$ as defined in §3 of Part I.

THEOREM 2.1. *For any ordinal $\iota \leq \epsilon_0$ and all integers $m \geq 0$,*

$$\sum_{n \geq 1, \alpha_\iota(n) = m} 2^{-L_\iota(n)} = 1.$$

Proof (by transfinite induction on ι). We have already proved, in the lemma in §1 of Part I, that this holds for $\iota < \omega$ and, in Theorem 3.1 of Part I, that it holds for $\iota = \omega$, so the base cases are done. When ι is a successor ordinal, the proof identically parallels that of the lemma in §1 of Part I; when ι is a limit ordinal, the proof identically parallels that of Theorem 3.1 of Part I. Since we will prove a more general form of this theorem in the next section (Theorem 3.1), we leave the details here to the reader. □

Remark. Theorem 2.1 can be rephrased as: for all ordinals $\iota \leq \epsilon_0$ and integers $m \geq 0$,

$$\sum_{n = A_\iota(m)}^{A_\iota(m+1)-1} 2^{-L_\iota(n)} = 1.$$

COROLLARY 2.2. *For any ordinal $\iota \leq \epsilon_0$ and integers $m \geq 1$,*

$$\alpha_\iota(m) \leq \sum_{n=1}^{m-1} 2^{-L_\iota(n)} < \alpha_\iota(m) + 1.$$

COROLLARY 2.3. *For any ordinal $\iota \leq \epsilon_0$, $\sum_{n=1}^{\infty} 2^{-L_\iota(n)}$ diverges.*

COROLLARY 2.4. *For any ordinal $\iota \leq \epsilon_0$, $L_\iota(n)$ is a lower bound on the unbounded searching problem of Bentley and Yao* [2] *and on prefix-free, binary encodings of the nonnegative integers* [4] *(see* [11]*), in the sense that if an unbounded search algorithm uses $c(n)$ probes to locate n (or, equivalently, a prefix-free code uses $c(n)$ bits to encode n) then $c(n) > L_\iota(n)$ for infinitely many n.*

Proof. This follows from Corollary 2.3 by Kraft's inequality (Part I, ineq. (1)). □

THEOREM 2.5. *For any ordinal $\iota \leq \epsilon_0$, there is an unbounded search algorithm in which the cost of finding n is*

$$L_\iota(n) + \alpha_\iota(n) + \begin{cases} 2 & n \leq 1, \\ 1 & n > 1. \end{cases}$$

Proof. The search algorithm is a unification of the level-by-level unbounded search in Algorithm 1 of Part I and the diagonal unbounded search in Algorithm 3 of Part I. We begin by finding the location of the last X relative to A_ι; that is, we find k such that

$$A_\iota(k) \leq \text{location of the last X} < A_\iota(k+1).$$

For $n > 1$, this costs $\alpha_\iota(n) + 1$ tests. Having found k, we apply either the level-by-level search or the diagonal search, depending whether ι is successor ordinal or a limit ordinal, respectively. Both these cases can be combined into a single function:

ALGORITHM 1

Ordinal searching, in pseudo-Pascal. For a successor ordinal ι, $\text{pred}(\iota)$ *is the predecessor of* ι *so that* $\iota = \text{pred}(\iota) + 1$. *For a limit ordinal* ι *of the sequence* $\{\iota_u\}$, $\text{pred}(\iota, u) = \iota_u$.

```
function OrdinalSearch(
    λ: ordinal; { level of the search }
    F: function(integer): [X, Y] { if F(n0) = Y then F(n) = Y for all n ≥ n0 }
    ): integer; { the location of the last X of F }
var
    k: integer; { location of the last X relative to Aλ; that is, Aλ(k) ≤ location < Aλ(k + 1) }

function OrdinalLevelSearch(
    ι: ordinal; { level of the search }
    u: integer { location of last X, relative to f(Aι) }
    f: function(integer): integer
    ): integer; { location of the last X, relative to f; that is, i such that f(i) ≤ location < f(i + 1) }
var
    j: integer; { when ι is a limit ordinal, j is computed to be the location of the last X relative
                  to f(A_pred(ι,u)); that is, f(A_pred(ι,u)(j)) ≤ location < f(A_pred(ι,u)(j + 1)) }
    found: boolean; { flag to terminate the while loop }
begin { OrdinalLevelSearch }
    { find the location of the last X relative to f, given that its location relative to f(Aι) is u;
      that is, it satisfies f(Aι(u)) ≤ location of the last X < f(Aι(u + 1)) }
    if u = 0 then { f(1) ≤ location of the last X < f(2) }
        return(1)
    else if ι = 0 then { f(2u) ≤ location < f(2u + 2) }
        if F(f(2u + 1)) = X then return(2u + 1) else return(2u)
    else if (ι is a successor ordinal) then { go down a level }
        return(OrdinalLevelSearch(pred(ι), OrdinalLevelSearch(ι, u − 1, f(A_pred(ι))), f))
    else begin { ι is a limit ordinal; compute the location of the last X relative to f(A_pred(ι,u)) }
        found := false;
        j := u;
        while (j < α_pred(ι,u)(Aι(u + 1) − 1)) and (not found) do
            if F(f(A_pred(ι,u)(j + 1))) = X then
                j := j + 1
            else { terminate the loop }
                found := true;
        return(OrdinalLevelSearch(pred(ι, u), j, f))
        end
end; { OrdinalLevelSearch }

begin { OrdinalSearch }
    k := 0;
    while F(Aλ(k + 1)) = X do
        k := k+1;
    { k > 0 is the location of the last X, relative to Aλ; that is, Aλ(k) ≤ location < Aλ(k + 1) }
    if k = 0 then { the location of the last X is either 0 or 1 }
        if F(1) = X then return(1) else return(0)
    else
        return(OrdinalLevelSearch(λ, k, f(x) = x))
end; { OrdinalSearch }
```

OrdinalLevelSearch, given in Algorithm 1. Transfinite induction verifies the correctness of *OrdinalLevelSearch* and that it requires $L_\iota(n)$ tests for $n > 1$. Thus the function *OrdinalSearch* in Algorithm 1 uses the stated number of tests. □

Corollary 2.4 tells us that the search procedure in Algorithm 1 is within an additive factor $\alpha_\iota(n) + 2$ of being optimal. For $\iota = 2$, this is equivalent to Bentley and Yao's result [2]; for $\iota = \omega$, this is the result of §3 of Part I on the diagonal algorithm. For each $\iota > \omega$, this is an enormous improvement because if $\tau > \sigma$ then $\alpha_\tau(n)$ is *much* more slowly growing than $\alpha_\sigma(n)$ as $n \to \infty$.

THEOREM 2.6. *For any ordinal $\iota \le \epsilon_0$, there is a uniquely decipherable, irredundant, lexicographic, prefix-free, binary code for the nonnegative integers in which the number of bits used to represent n is*

$$L_\iota(n) + \alpha_\iota(n) + \begin{cases} 2 & n \le 1, \\ 1 & n > 1. \end{cases}$$

Proof. We define

$$B_0(n) = \begin{cases} \text{empty string} & n = 1, \\ n \bmod 2 & n > 1; \end{cases}$$

and for successor ordinals ι, let

$$B_\iota(n) = \begin{cases} \text{empty string} & n = 1, \\ B_\iota(\alpha_{\text{pred}(\iota)}(n)) B_{\text{pred}(\iota)}(n) & n > 1; \end{cases}$$

and for limit ordinals,

$$B_\iota(n) = \begin{cases} \text{empty string} & \text{if } n = 1, \\ 1^{\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n)} B_{\text{pred}(\iota,\alpha_\iota(n))}(n) & \text{if } \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) = \\ & \quad \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(A_\iota(\alpha_\iota(n)+1) - 1), \\ 1^{\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n)} 0 B_{\text{pred}(\iota,\alpha_\iota(n))}(n) & \text{otherwise.} \end{cases}$$

Now we can define the code

$$C_\iota(n) = \begin{cases} 00 & \text{if } n = 0, \\ 01 & \text{if } n = 1, \\ 1^{\alpha_\iota(n)} 0 B_\iota(n) & \text{if } n > 1, \end{cases}$$

whose length is as stated. Clearly, our definition of B_ι is consistent with that given for B_i in §2 of Part I; C_ω is precisely C, as defined in §3 of Part I.

The decoding algorithm for C_ι is given in Algorithm 2, which is little more than a combination of Algorithms 2 and 4 of Part I. The correctness of Algorithm 2 follows by transfinite induction, along with the observation that whenever the function *DecodeOrdinalB* is called to decode $B_\iota(n)$, the parameter $prefix$ has the value $\alpha_\iota(n)$. The nature of the decoding procedure guarantees that C_ι is prefix-free as well as uniquely decipherable.

As in the case of C_i and C, the irredundancy is a consequence of the appropriate sum being equal to one; for C_ι this breaks into two cases—when ι is a successor ordinal, and when it is a limit ordinal. The lexicographic property is similarly apparent. □

3. Extension to asymmetric costs. Let $x > 0$ and $y > 0$; define the weighted length of $B_\iota(n)$ to be

$$W_\iota(n) = x \times [\text{number of one-bits in } B_\iota(n)] + y \times [\text{number of zero-bits in } B_\iota(n)].$$

ALGORITHM 2

Decoding the codes in the ordinal construction, in pseudo-Pascal. For a successor ordinal ι, $\mathrm{pred}(\iota)$ *is the predecessor of* ι *so that* $\iota = \mathrm{pred}(\iota)+1$. *For a limit ordinal* ι *of the sequence* $\{\iota_u\}$, $\mathrm{pred}(\iota, u) = \iota_u$.

```
function DecodeOrdinal(
    λ: ordinal; { level of the code }
    b: BitString { the codeword to be decoded }
    ): integer; { the decoded value }
var
    k: integer; { index of last bit of b processed }
    p: integer; { value of prefix part of b }

function DecodeOrdinalB(
    ι: ordinal; { level of the code }
    prefix: integer { prefix value }
    ): integer; { decoded value }
var
    j: integer; { diagonal index value, when ι is a limit ordinal }
    found: boolean; { flag to terminate the while loop }
begin { DecodeOrdinalB }
    if prefix = 0 then { the empty string encodes 1 }
        return(1)
    else if ι = 0 then begin { value encoded is twice the prefix with last bit added }
        k := k + 1;
        return(2 * prefix + b_k)
        end
    else if (ι is a successor ordinal) then
        { decode the next prefix and use it to decode the remaining bits }
        return(DecodeOrdinalB(pred(ι), DecodeOrdinalB(ι, prefix − 1)))
    else begin {ι is a limit ordinal }
        found := false;
        j := prefix;
        while (j < α_pred(ι,prefix)(A_ι(prefix + 1) − 1)) and (not found) do
            if b_{k+1} = 1 then begin
                k := k + 1;
                j := j + 1
                end
            else { terminate the loop }
                found := true;
        if found then
            k := k + 1; { skip the following 0 }
        return(DecodeOrdinalB(pred(ι,prefix), j))
        end
end; { DecodeOrdinalB }

begin { DecodeOrdinal }
    k := 0;
    while b_{k+1} = 1 do
        k := k+1;
    { k = α_λ(n); b_1 = ··· = b_k = 1 and b_{k+1} = 0 }
    p := k;
    k := k + 1; { skip the 0 }
    if p = 0 then
        if b_2 = 1 then return(1) else return(0)
    else
        return(DecodeOrdinalB(λ, p))
end; { DecodeOrdinal }
```

Then, obviously, for any ordinal $\iota \leq \epsilon_0$, there is a uniquely decipherable, irredundant, lexicographic, prefix-free, binary code for the nonnegative integers in which the weighted cost of the representation of n is

$$W_\iota(n) + x\alpha_\iota(n) + \begin{cases} 2y & n = 0, \\ x + y & n = 1, \\ y & n > 1. \end{cases}$$

Therefore, for any ordinal $\iota \leq \epsilon_0$, there is an unbounded search algorithm in which the weighted cost of finding n is

$$W_\iota(n) + x\alpha_\iota(n) + \begin{cases} 2y & n = 0, \\ x + y & n = 1, \\ y & n > 1. \end{cases}$$

THEOREM 3.1. *For any ordinal $\iota \leq \epsilon_0$ and all integers $m \geq 0$,*

$$\sum_{n \geq 1, \alpha_\iota(n) = m} r^{-W_\iota(n)} = 1.$$

Proof (by transfinite induction on ι). For the basis we need to consider the case $m = 0$ and the case $\iota = 0$, $m > 0$. When $m = 0$, $\alpha_\iota(n) = 0$ only for $n \leq 1$; $B_\iota(1)$ is empty so $W_\iota(1) = 0$ and the sum is correct. When $\iota = 0$, $\alpha_0(n) = m$ for $n = 2m$ or $n = 2m+1$; $B_0(2m) = 0$ and $B_0(2m+1) = 1$ giving $W_0(2m) = y$ and $W_0(2m+1) = x$, so the sum is simply $r^{-x} + r^{-y} = 1$ by the definition of r.

Suppose ι is a successor ordinal and $m > 0$. In this case $n > 1$ so $B_\iota(n) = B_\iota(\alpha_{\text{pred}(\iota)}(n))B_{\text{pred}(\iota)}(n)$ making $W_\iota(n) = W_\iota(\alpha_{\text{pred}(\iota)}(n)) + W_{\text{pred}(\iota)}(n)$. Following the proof of the lemma in §1 of Part I [15], observe that by the definition of $\alpha_\iota(n)$ $\alpha_\iota(\alpha_{\text{pred}(\iota)}(n)) = \alpha_\iota(n) - 1$, so that by the definition of $W_\iota(n)$

$$\begin{aligned} \sum_{n \geq 1, \alpha_\iota(n) = m} r^{-W_\iota(n)} &= \sum_{n \geq 1, \alpha_\iota(\alpha_{\text{pred}(\iota)}(n)) = m-1} r^{-W_{\text{pred}(\iota)}(n) - W_\iota(\alpha_{\text{pred}(\iota)}(n))} \\ &= \sum_{k \geq 1, \alpha_\iota(k) = m-1} r^{-W_\iota(k)} \sum_{n \geq 1, \alpha_{\text{pred}(\iota)}(n) = k} r^{-W_{\text{pred}(\iota)}(n)}, \end{aligned}$$

by substituting k for $\alpha_{\text{pred}(\iota)}(n)$ and rewriting. But by induction the inner sum is 1, giving

$$\begin{aligned} &= \sum_{k \geq 1, \alpha_\iota(k) = m-1} r^{-W_\iota(k)} \\ &= 1, \end{aligned}$$

again by induction.

Finally, when $\iota > 0$ is a limit ordinal and $m > 0$, let us simplify the notation by writing ψ for pred(ι, m) and letting $\alpha_\psi(A_\iota(\alpha_\iota(n) + 1) - 1) = m + h$. Then from the definition of $B_\iota(n)$,

$$W_\iota(n) = \begin{cases} x \times [\alpha_\psi(n) - m] + W_\psi(n) & \text{if } \alpha_\psi(n) = m + h, \\ x \times [\alpha_\psi(n) - m] + y + W_\psi(n) & \text{otherwise.} \end{cases}$$

Then, as in the proof of Theorem 3.1 in Part I, the sum becomes

$$\begin{aligned}\sum_{n\geq 1,\alpha_\iota(n)=m} r^{-W_\iota(n)} &= \sum_{j=0}^{h-1} \sum_{n\geq 1,\alpha_\psi(n)=m+j} r^{-jx-y-W_\psi(n)} \\ &\qquad + \sum_{n\geq 1,\alpha_\psi(n)=m+h} r^{-hx-W_\psi(n)} \\ &= \sum_{j=0}^{h-1} r^{-jx-y} + r^{-hx} \\ &= 1. \end{aligned}$$ □

COROLLARY 3.2. *For any ordinal $\iota \leq \epsilon_0$, $W_\iota(n)$ is a lower bound on the asymmetric-cost version of the unbounded searching problem of Bentley and Yao* [2] *and on the asymmetric-cost version of prefix-free, binary encodings of the nonnegative integers* [4] *(see* [11]*), in the sense that if an unbounded search algorithm uses cost $c(n)$ to locate n (or, equivalently, a prefix-free code uses cost $c(n)$ to encode n) then $c(n) > W_\iota(n)$ for infinitely many n.*

Since $\sum_{n=1}^{\infty} r^{-\log_r n}$ diverges, $\log_r n$ is a lower bound for the asymmetric case by the generalized Kraft's inequality (Part I, ineq. (5)). Thus we are not interested in lower bounds below $\log_r n$, just as in the symmetric case we are not interested in lower bounds below $\lg n$.

Our level ι algorithm/encoding, as presented so far, is within about $x\alpha_\iota(n)$ of being optimal in the original sense of [2], but it is *not* that close to being optimal in the stronger sense defined in §5 of Part I. Specifically,

$$\limsup_{n\to\infty} \frac{W_\iota(n)}{\log_r n} = \max\{x, y\} \lg r$$

by an argument identical to that in the case $\iota < \omega$ given in Part I. Thus

$$\limsup_{n\to\infty} \frac{W_\iota(n)}{\log_r n}$$

is finite for any choice of x and y, but it can be arbitrarily large. In other words, our level ι algorithm/encoding is within a constant *multiplicative* factor of being optimal in the asymmetric case, not within a slowly growing *additive* factor.

As discussed in §6 of Part I, to obtain such a near-optimal algorithm/encoding for the asymmetric case, we must redefine $A_0(n)$ and $\alpha_0(n)$ as

$$A_0(n) = \lceil r^y n \rceil,$$

$$\alpha_0(n) = \lfloor n/r^y \rfloor;$$

the recursive definitions of $A_\iota(n)$ and $\alpha_\iota(n)$ for $\iota > 0$ are unchanged. The algorithm is essentially identical to Algorithm 1, but with modifications similar to those needed in the finite case (§6 of Part I). As there, we define

$$L_0(0) = 0,$$

and for $n \geq 1$,

$$L_0(n) = \begin{cases} \text{the exact cost to search for } n \text{ in the interval} \\ [A_0(\alpha_0(n)), A_0(\alpha_0(n)+1)-1] \text{ using the op-} \\ \text{timum bounded lopsided search algorithm} \\ \text{of [9, §2.2].} \end{cases}$$

Generalizing Part I, we define

$$L_\iota(n) = \begin{cases} 0 & n = 0, \\ L_0(n) & 1 \leq n < \lceil r^y \rceil, \\ L_{\text{pred}(\iota)}(n) + L_\iota(\alpha_{\text{pred}(\iota)}(n)) & n \geq \lceil r^y \rceil, \end{cases}$$

for successor ordinals, and

$$L_\iota(n) = \begin{cases} 0 & n = 0, \\ L_0(n) & 1 \leq n < \lceil r^y \rceil, \\ L_{\text{pred}(\iota,\alpha_\iota(n))}(n) & n \geq \lceil r^y \rceil \text{ and } \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) = \\ \quad +x(\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n)) & \quad \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(A_\iota(\alpha_\iota(n)+1)-1), \\ L_{\text{pred}(\iota,\alpha_\iota(n))}(n) & \\ \quad +x(\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n)) + y & \text{otherwise,} \end{cases}$$

for limit ordinals. Then a transfinite induction similar to the proof of Theorem 3.1 shows that for $\iota \geq 0$,

$$\sum_{\alpha_\iota(n)=m} r^{-L_\iota(n)} = 1,$$

making the $L_\iota(n)$ lower bounds on the asymmetric unbounded search problem. Techniques parallel to those in the next section prove that, for $1 \leq \iota < \epsilon_0$,

$$L_\iota(n) = \log_r n + O(\lg \lg n),$$

so that these lower bounds are nontrivial. The corresponding algorithms/encodings have cost

$$L_\iota(n) + x\alpha_\iota(n) + \begin{cases} 2y & n = 0, \\ x + y & 1 \leq n < \lceil r^y \rceil, \\ y & n \geq \lceil r^y \rceil, \end{cases}$$

and are thus within an additive factor $x\alpha_\iota(n) + \max\{2y, x+y\}$ of being optimal in the strong sense of §5 of Part I.

4. The asymptotic behavior of $L_\iota(n)$. We show in this section that $L_\iota(n) = \lg n + O(\lg \lg n)$ for $1 \leq \iota < \epsilon_0$; this implies that

$$\limsup_{n\to\infty} \frac{L_\iota(n)}{b(n)} = 1$$

for any nontrivial lower bound $b(n)$ and thus all the algorithms in §2 are optimal in a strong sense. Verifying this seemingly simple statement about $L_\iota(n)$ requires a surprising effort! The difficulty is the diagonalization, together with the addition of 1, under certain conditions, in the recurrence relation for $L_\iota(n)$ when ι is a limit ordinal. The functions $L_\iota(n)$ cannot be written in closed form, and they are not monotonic in any simple way. Since more direct techniques fail to work, we approach the problem by computing a separate bound on the number of diagonalizations that occur in the

evaluation of $L_\iota(n)$, and then using that bound in studying the growth of $L_\iota(n)$. The number of diagonalizations is given by the function $d_\iota(n)$, defined by the following definition.

DEFINITION. For ordinals $\iota \le \epsilon_0$ the *depth function* $d_\iota(n)$ is defined for positive integers n by

$$d_\iota(n) = \begin{cases} 0 & \text{if } \iota = 0, \\ d_{\text{pred}(\iota)}(n) & \text{if } \iota \text{ is a successor ordinal,} \\ d_{\text{pred}(\iota,\alpha_\iota(n))}(n) + 1 & \text{if } \iota \text{ is a limit ordinal.} \end{cases}$$

For example, $d_i(n) = 0$ for nonnegative integers i; $d_\omega(n) = 1$ for all n; $d_{\omega\cdot 2}(n) = 2$ for all n; $d_{\omega\cdot i+k}(n) = i$ for all nonnegative integers i, k, and n; $d_{\omega^2}(n) = \alpha_{\omega^2}(n) + 1$. Our goal is to prove that $d_\iota(n)$ is extremely small, compared to n. We will prove that for all ordinals $\iota < \epsilon_0$, $d_\iota(n)$ is majorized by $\alpha_i(n)$ for all integers $i \ge 0$.

To facilitate the study of $d_\iota(n)$ and $L_\iota(n)$, we need to know when an ordinal σ occurs as one of the predecessors in the sequence of diagonalizations leading to $L_\iota(n)$. We use a generalization of a concept and notation from [10], who needed similar machinery in their study of $A_\iota(n)$.

DEFINITION. Let $g : \{\iota \mid \iota \le \epsilon_0\} \to N^+$ be a positive-integer-valued function of the ordinals up to ϵ_0. We say an ordinal σ *traces to* an ordinal τ *under* g, written $\sigma \xrightarrow{g} \tau$, if there is a finite sequence of ordinals $\gamma_0, \gamma_1, \cdots, \gamma_r$ such that $\gamma_0 = \sigma$, $\gamma_r = \tau$, and

$$\gamma_{i+1} = \begin{cases} \text{pred}(\gamma_i) & \text{if } \gamma_i \text{ is a successor ordinal,} \\ \text{pred}(\gamma_i, g(\gamma_i)) & \text{if } \gamma_i \text{ is a limit ordinal.} \end{cases}$$

For example, if $g(\iota) = n$ for a fixed positive integer n, $\omega^2 \xrightarrow{g} \omega \cdot n \xrightarrow{g} \omega \xrightarrow{g} 0$. This particular form of tracing was used in [10] in their analysis of the generalized Ackermann's function. As another example, note that ι traces to $\text{pred}(\iota, \alpha_\iota(n))$, under $g(\iota) = \alpha_\iota(n)$ for a fixed positive integer n; this form of tracing will be of importance to us in our analysis of $L_\iota(n)$.

DEFINITION. If for all functions $g : \{\iota \mid \iota \le \epsilon_0\} \to N^+$ we have $\sigma \xrightarrow{g} \tau$, then we say that σ *traces to* τ *absolutely*, written $\sigma \xrightarrow{\forall} \tau$.

It is not difficult to verify, for example, that $\tau + 1 \xrightarrow{\forall} \tau$ for any $\tau < \epsilon_0$, or that $\omega \cdot 2 \xrightarrow{\forall} \omega \xrightarrow{\forall} 1 \xrightarrow{\forall} 0$.

LEMMA 4.1. *For all limit ordinals $\iota \le \epsilon_0$ and all nonnegative integers j and k, $j < k$, $\text{pred}(\iota, k) \xrightarrow{\forall} \text{pred}(\iota, j)$.*

The proof of Lemma 4.1, given below, relies on a number of propositions. Our presentation of these propositions closely parallels the proof of Theorem 2.4 in [10], but our Lemma 4.1 is a much stronger result, from which Theorem 2.4 in [10] follows as a corollary. The following four propositions are evident from the above definitions.

PROPOSITION 4.2. *If $\tau_1 \xrightarrow{g} \tau_2$, $\tau_1 \xrightarrow{g} \tau_3$, and $\tau_2 > \tau_3$, then $\tau_2 \xrightarrow{g} \tau_3$.*

PROPOSITION 4.3. *If $\tau_1 \xrightarrow{g} \tau_2$ and $\tau_2 \xrightarrow{g} \tau_3$, then $\tau_1 \xrightarrow{g} \tau_3$.*

PROPOSITION 4.4. *If $\tau_1 \xrightarrow{\forall} \tau_2$, $\tau_1 \xrightarrow{\forall} \tau_3$, and $\tau_2 > \tau_3$, then $\tau_2 \xrightarrow{\forall} \tau_3$.*

PROPOSITION 4.5. *If $\tau_1 \xrightarrow{\forall} \tau_2$ and $\tau_2 \xrightarrow{\forall} \tau_3$, then $\tau_1 \xrightarrow{\forall} \tau_3$.*

DEFINITION (see, for example, [10], [12], or [16]). Any limit ordinal $\iota < \epsilon_0$ can be written in *Cantor normal form* as

$$\iota = \omega^{\beta_1} \cdot a_1 + \omega^{\beta_2} \cdot a_2 + \cdots + \omega^{\beta_r} \cdot a_r,$$

where $\iota > \beta_1 > \beta_2 > \cdots > \beta_r > 0$ are ordinals and the a_i are natural numbers, $a_r > 0$.

DEFINITION (from [10]). Let $\sigma < \epsilon_0$ and $\tau < \epsilon_0$ be ordinals. We say that σ *meshes with* τ if for some ordinals $\eta > 0$ and $\beta > 0$ $\sigma = \omega^\eta \cdot \beta$ and $\tau < \omega^{\eta+1}$.

This definition means that two ordinals "mesh" if their Cantor normal forms concatenate to form the Cantor normal form of their sum. Specifically, if σ meshes with τ,

$$\sigma = \omega^{\theta_1} \cdot n_1 + \cdots + \omega^{\theta_k} \cdot n_k$$

$$\tau = \omega^{\eta_1} \cdot m_1 + \cdots + \omega^{\eta_l} \cdot m_l,$$

then $\theta_k \geq \eta_1$. The notion of meshing is helpful because the pred function concentrates its action on the rightmost (smallest) term in the Cantor normal form of an ordinal less than ϵ_0. As a consequence, for instance, the following proposition.

PROPOSITION 4.6. *Given ordinals $\sigma < \epsilon_0$ and $\tau < \epsilon_0$ such that σ meshes with τ, then for any nonnegative integer n,* $\mathrm{pred}(\sigma + \tau, n) = \sigma + \mathrm{pred}(\tau, n)$, *and hence if $\tau \xrightarrow{\forall} \beta$ for some ordinal β then $(\sigma + \tau) \xrightarrow{\forall} (\sigma + \beta)$.*

Proof. Because $\sigma + \tau < \epsilon_0$, the meshing of σ and τ, taken with the way $\mathrm{pred}(\sigma + \tau, n)$ differs from $\sigma + \tau$ only in the smallest term of the Cantor normal form of $\sigma + \tau$, guarantees that

$$\mathrm{pred}(\sigma + \tau, n) = \sigma + \mathrm{pred}(\tau, n)$$

for any nonnegative integer n, as stated in the proposition. Then, given any positive-integer-valued function g on the ordinals, we define $\hat{g}(\iota) = g(\sigma + \iota)$. We have

$$\begin{aligned} \mathrm{pred}(\sigma + \tau, g(\sigma + \tau)) &= \sigma + \mathrm{pred}(\tau, g(\sigma + \tau)) \\ &= \sigma + \mathrm{pred}(\tau, \hat{g}(\tau)). \end{aligned}$$

Therefore, if $\tau \xrightarrow{\hat{g}} \beta$, then $(\sigma + \tau) \xrightarrow{g} (\sigma + \beta)$. But by hypothesis $\tau \xrightarrow{\forall} \beta$ and hence $\tau \xrightarrow{\hat{g}} \beta$ for any choice of $\hat{g}$; that is, for any choice of g, $(\sigma + \tau) \xrightarrow{g} (\sigma + \beta)$, which means $(\sigma + \tau) \xrightarrow{\forall} (\sigma + \beta)$. □

PROPOSITION 4.7. *For all ordinals $\iota \leq \epsilon_0$, $\iota \xrightarrow{\forall} 0$.*

Proof. A trivial transfinite induction on ι. □

PROPOSITION 4.8. *For all ordinals $\iota < \epsilon_0$ and all integers $k > j > 0$,*

$$\omega^\iota \cdot k \xrightarrow{\forall} \omega^\iota \cdot j.$$

Proof. By Proposition 4.7, $\omega^\iota \cdot (k - j) \xrightarrow{\forall} 0$. Also, $\omega^\iota \cdot j$ meshes with $\omega^\iota \cdot (k - j)$, so Proposition 4.6 applies to yield $\omega^\iota \cdot j + \omega^\iota \cdot (k - j) \xrightarrow{\forall} \omega^\iota \cdot j + 0$; that is, $\omega^\iota \cdot k \xrightarrow{\forall} \omega^\iota \cdot j$, as desired. □

PROPOSITION 4.9. *For all ordinals $\iota < \epsilon_0$, $\omega^{\iota+1} \xrightarrow{\forall} \omega^\iota$.*

Proof. For any integer n,

$$\mathrm{pred}(\omega^{\iota+1}, n) = \omega^\iota \cdot n,$$

so that for any positive-integer-valued function of the ordinals g,

$$\mathrm{pred}(\omega^{\iota+1}, g(\omega^{\iota+1})) = \omega^{\iota} \cdot g(\omega^{\iota+1}).$$

If $g(\omega^{\iota+1}) = 1$, we are done; if not, from Proposition 4.8,

$$\omega^{\iota} \cdot g(\omega^{\iota+1}) \xrightarrow{\forall} \omega^{\iota},$$

so in particular,

$$\omega^{\iota} \cdot g(\omega^{\iota+1}) \xrightarrow{g} \omega^{\iota}.$$

Hence, by Proposition 4.3,

$$\omega^{\iota+1} \xrightarrow{g} \omega^{\iota} \cdot g(\omega^{\iota+1}) \xrightarrow{g} \omega^{\iota}.$$

But g was arbitrary, so $\omega^{\iota+1} \xrightarrow{\forall} \omega^{\iota}$. □

PROPOSITION 4.10. *Let* $\eta_1 < \epsilon_0$ *and* $\eta_1 \xrightarrow{\forall} \eta_2$*; then* $\omega^{\eta_1} \xrightarrow{\forall} \omega^{\eta_2}$.

Proof. The proof is by induction on η_1. If $\eta_1 < \omega$, the desired result holds by Proposition 4.9. For $\eta_1 \geq \omega$, η_1 a successor ordinal, the proposition follows from Proposition 4.9 and induction. For $\eta_1 \geq \omega$, η_1 a limit ordinal, for any positive-integer-valued function g on the ordinals,

$$\mathrm{pred}(\omega^{\eta_1}, g(\omega^{\eta_1})) = \omega^{\mathrm{pred}(\eta_1, g(\omega^{\eta_1}))}.$$

If $\mathrm{pred}(\eta_1, g(\omega^{\eta_1})) = \eta_2$, we are done; if not,

$$\mathrm{pred}(\eta_1, g(\omega^{\eta_1})) \xrightarrow{\forall} \eta_2$$

since $\eta_1 \xrightarrow{\forall} \eta_2$, so by induction,

$$\mathrm{pred}(\omega^{\eta_1}, g(\omega^{\eta_1})) = \omega^{\mathrm{pred}(\eta_1, g(\omega^{\eta_1}))} \xrightarrow{\forall} \omega^{\eta_2}.$$

Specifically, then $\omega^{\eta_1} \xrightarrow{g} \omega^{\eta_2}$, and the desired result follows because g was arbitrary. □

Proof of Lemma 4.1 (by transfinite induction on ι). If $\iota = \omega$,

$$\mathrm{pred}(\iota, k) = k \xrightarrow{\forall} j = \mathrm{pred}(\iota, j),$$

establishing the basis. For the induction itself, there are four cases, depending on the form of ι.

Case 1. $\iota = \omega^{\eta} \cdot (\beta + 1)$, $\beta > 0$. We have

$$\mathrm{pred}(\iota, k) = \omega^{\eta} \cdot \beta + \mathrm{pred}(\omega^{\eta}, k),$$

so by induction,

$$\mathrm{pred}(\omega^{\eta}, k) \xrightarrow{\forall} \mathrm{pred}(\omega^{\eta}, j),$$

and hence, by Proposition 4.6, $\mathrm{pred}(\iota, k) \xrightarrow{\forall} \mathrm{pred}(\iota, j)$.

Case 2. $\iota = \omega^{\eta+1}$. Here,

$$\text{pred}(\iota, k) = \omega^{\eta} \cdot k \overset{\forall}{\rightarrow} \omega^{\eta} \cdot j = \text{pred}(\iota, j)$$

by Proposition 4.8.

Case 3. $\iota = \omega^{\eta}$, η a limit ordinal. In this case we have

$$\text{pred}(\iota, k) = \omega^{\text{pred}(\eta,k)}$$

and

$$\text{pred}(\iota, j) = \omega^{\text{pred}(\eta,j)}.$$

By induction $\text{pred}(\eta, k) \overset{\forall}{\rightarrow} \text{pred}(\eta, j)$ and by Proposition 4.10 $\text{pred}(\iota, k) \overset{\forall}{\rightarrow} \text{pred}(\iota, j)$.

Case 4. $\iota = \epsilon_0$. Here, $\text{pred}(\iota, k)$ is a k-high tower of omegas and $\text{pred}(\iota, j)$ is a j-high tower of omegas. Since $\omega \overset{\forall}{\rightarrow} 1$, by Proposition 4.10 $\omega^{\omega} \overset{\forall}{\rightarrow} \omega$. Applying this $k-1$ times yields $\text{pred}(\iota, k) \overset{\forall}{\rightarrow} \text{pred}(\iota, k-1)$, which implies $\text{pred}(\iota, k) \overset{\forall}{\rightarrow} \text{pred}(\iota, j)$. □

Theorem 2.4 and its corollary from [10] follow as corollaries to Lemma 4.1. There are several other useful and interesting corollaries, too.

COROLLARY 4.11. *For any fixed nonnegative integer n, let* $g(\iota) = \alpha_{\iota}(n)$*; then for all limit ordinals* $\iota < \epsilon_0$ *and all integers* $k > 0$, $\text{pred}(\iota, k+1) \overset{g}{\rightarrow} \text{pred}(\iota, k)$.

COROLLARY 4.12. L_{ι} *and* d_{ι} *are "monotonic" in the sense that for all limit ordinals* $\iota < \epsilon_0$*, and for all nonnegative integers k and n,*

$$L_{\text{pred}(\iota,k+1)}(n) \geq L_{\text{pred}(\iota,k)}(n), \tag{1}$$

$$d_{\text{pred}(\iota,k+1)}(n) \geq d_{\text{pred}(\iota,k)}(n). \tag{2}$$

Proof. These inequalities follow from Lemma 4.1 and the formulas for $L_{\iota}(n)$ and $d_{\iota}(n)$. □

COROLLARY 4.13. *For any ordinal* $\iota < \epsilon_0$, $d_{\iota}(n) \geq d_{\iota}(m)$ *if* $n > m$.

Proof. The proof is by transfinite induction. For $\iota \leq \omega$, the result is obvious. Suppose it is true for all ordinals less than ι. If ι is a successor ordinal, $d_{\iota}(n) = d_{\text{pred}(\iota)}(n)$ and $d_{\iota}(m) = d_{\text{pred}(\iota)}(m)$, so the result holds by induction.

If ι is a limit ordinal then

$$d_{\iota}(n) = d_{\text{pred}(\iota,\alpha_{\iota}(n))}(n) + 1$$

and

$$d_{\iota}(m) = d_{\text{pred}(\iota,\alpha_{\iota}(m))}(m) + 1.$$

If $\alpha_{\iota}(n) = \alpha_{\iota}(m)$, then by induction

$$d_{\text{pred}(\iota,\alpha_{\iota}(n))}(n) \geq d_{\text{pred}(\iota,\alpha_{\iota}(m))}(m)$$

and the result holds. On the other hand, if $\alpha_{\iota}(n) > \alpha_{\iota}(m)$ then by Lemma 4.1 $\text{pred}(\iota, \alpha_{\iota}(n)) \overset{\forall}{\rightarrow} \text{pred}(\iota, \alpha_{\iota}(m))$ and hence by inequality (2)

$$d_{\text{pred}(\iota,\alpha_{\iota}(n))}(m) \geq d_{\text{pred}(\iota,\alpha_{\iota}(m))}(m),$$

and the result follows. □

COROLLARY 4.14. *For all ordinals $\iota < \epsilon_0$, if $\alpha_\iota(n+1) = \alpha_\iota(n) + 1$, then*

$$\alpha_{\text{pred}(\iota)}(n+1) = \alpha_{\text{pred}(\iota)}(n) + 1$$

for ι a successor ordinal, while

$$\alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n+1) = \alpha_{\text{pred}(\iota,\alpha_\iota(n))}(n) + 1$$

for ι a limit ordinal.

Proof. For convenience, let $\alpha_\iota(n) = m$ and $\alpha_\iota(n+1) = m+1$. If ι is a successor ordinal, say $\iota = \beta + 1$, then

$$A_{\beta+1}(m) \le n < A_{\beta+1}(m+1),$$

since $\alpha_\iota(n) = m$, while

$$A_{\beta+1}(m+1) \le n+1 < A_{\beta+1}(m+2),$$

since $\alpha_\iota(n+1) = m+1$. These two inequalities imply that

$$A_{\beta+1}(m+1) = n+1,$$

however,

$$A_{\beta+1}(m+1) = A_\beta^{(m+1)}(1) = A_\beta(A_\beta^{(m)}(1)) = A_\beta(A_{\beta+1}(m)),$$

so

$$A_\beta(A_{\beta+1}(m)) = n+1.$$

Applying α_β to both sides of this equation yields

$$A_{\beta+1}(m) = \alpha_\beta(n+1).$$

The penultimate equation means that $n+1$ is the least number x such that $\alpha_\beta(x) \ge A_{\beta+1}(m)$; therefore

$$\alpha_\beta(n) < A_{\beta+1}(m).$$

That is,

$$\alpha_\beta(n) < A_{\beta+1}(m) = \alpha_\beta(n+1),$$

or,

$$\alpha_{\text{pred}(\iota)}(n+1) > \alpha_{\text{pred}(\iota)}(n).$$

But clearly,

$$\alpha_{\text{pred}(\iota)}(n+1) \le \alpha_{\text{pred}(\iota)}(n) + 1$$

and the result follows.

If ι is a limit ordinal, we have, in a manner parallel to the successor ordinal case,

$$A_\iota(m) \le n < A_\iota(m+1) \le n+1 < A_\iota(m+2).$$

Or equivalently,

$$A_{\text{pred}(\iota,m)}(m) \le n < A_{\text{pred}(\iota,m+1)}(m+1) \le n+1 < A_{\text{pred}(\iota,m+2)}(m+2).$$

Thus

$$A_{\text{pred}(\iota,m+1)}(m+1) = n+1,$$

meaning

$$\alpha_{\text{pred}(\iota,m+1)}(n+1) = m+1,$$

and

$$\alpha_{\text{pred}(\iota,m+1)}(n) = m;$$

that is, the corollary holds for $\text{pred}(\iota, m+1)$. In other words, the desired property holds for the ordinal $\text{pred}(\iota, \alpha_\iota(n)+1)$, a single tracing step down from ι under the function $g(\sigma) = \alpha_\sigma(n)+1$; that is, a tracing step under g preserves the property. However, we know from Lemma 4.1 that $\text{pred}(\iota, \alpha_\iota(n)+1) \xrightarrow{\forall} \text{pred}(\iota, \alpha_\iota(n))$, so that in particular $\text{pred}(\iota, \alpha_\iota(n)+1) \xrightarrow{g} \text{pred}(\iota, \alpha_\iota(n))$, and we have just seen that each individual tracing step under g preserves the needed property. □

Remark. This last corollary means that whenever there is a "jump" in the value of $\alpha_\iota(n)$, all of its predecessors have a "jump" in value also.

LEMMA 4.15. *If $\sigma < \epsilon_0$ meshes with $\tau < \epsilon_0$ then*

$$d_{\sigma+\tau}(n) \le d_\sigma(n) + d_\tau(n).$$

Proof. The proof is by transfinite induction on τ. If $\tau < \omega$, then $\tau = k$ for some integer $k \ge 0$. In this case

$$d_{\sigma+\tau}(n) = d_{\sigma+k}(n) = d_{\sigma+k-1}(n) = \cdots = d_\sigma(n),$$

so

$$d_{\sigma+\tau}(n) = d_\sigma(n) + d_\tau(n),$$

because $d_\tau(n) = 0$ by definition.

If τ is a successor ordinal, then

$$\begin{aligned} d_{\sigma+\tau}(n) &= d_{\text{pred}(\sigma+\tau)}(n) \\ &= d_{\sigma+\text{pred}(\tau)}(n) \end{aligned}$$

by Proposition 4.6 because σ meshes with τ;

$$\le \quad d_\sigma(n) + d_{\text{pred}(\tau)}(n)$$

by induction; and

$$= \quad d_\sigma(n) + d_\tau(n)$$

by the definition of $d_\tau(n)$.

If τ is a limit ordinal, then

$$d_{\sigma+\tau}(n) \quad = \quad d_{\sigma+\mathrm{pred}(\tau,\alpha_{\sigma+\tau}(n))}(n)+1$$

by definition and Proposition 4.6;

$$\leq \quad d_\sigma(n)+d_{\mathrm{pred}(\tau,\alpha_{\sigma+\tau}(n))}(n)+1$$

by induction;

$$\leq \quad d_\sigma(n)+d_{\mathrm{pred}(\tau,\alpha_{\tau}(n))}(n)+1$$

by inequality (2) since $\alpha_{\sigma+\tau}(n) \leq \alpha_\tau(n)$, as is easily verified; and

$$\leq \quad d_\sigma(n)+d_\tau(n)$$

by definition. □

Cantor normal form allows us to view any ordinal ι as a function $f_\iota(\omega)$, formed by additions, multiplications, and exponentiations of ordinals and nonnegative integers. If we consider the integer function $f_\iota(n)$ in which each instance of ω in $f_\iota(\omega)$ is replaced by n, then it is easily shown by transfinite induction that for successor ordinals ι,

$$f_\iota(n) = f_{\mathrm{pred}(\iota)}(n)+1, \tag{3}$$

while for limit ordinals ι, when $n \geq 1$, $k \leq n$, then

$$f_\iota(n) \geq f_{\mathrm{pred}(\iota,k)}(n). \tag{4}$$

LEMMA 4.16. *For any integer $n \geq 1$, and for any ordinal η, $1 \leq \eta < \epsilon_0$,*

$$\alpha_{\omega^\eta}(n) \leq \alpha_\eta(n).$$

Proof. It suffices to show that for $n \geq 1$,

$$A_{\omega^\eta}(n) \geq A_\eta(n),$$

because otherwise, let

$$u = \alpha_{\omega^\eta}(n^*) > \alpha_\eta(n^*) = v$$

for some n^*, and suppose that $A_{\omega^\eta}(n) \geq A_\eta(n)$ for any $n \geq 1$. Then $u \geq v+1$, so by the definition of the generalized inverse Ackermann's functions,

$$A_{\omega^\eta}(u) \leq n^* < A_{\omega^\eta}(u+1)$$

and

$$A_\eta(v) \le n^* < A_\eta(v+1),$$

giving

$$n^* < A_\eta(v+1) \le A_\eta(u) \le A_{\omega^\eta}(u) \le n^*,$$

a contradiction.

We will prove $A_{\omega^\eta}(n) \ge A_\eta(n)$ by transfinite induction on η. For $\eta = 1$, $\omega^\eta = \omega$ and

$$A_{\omega^\eta}(n) = A(n) = A_n(n) \ge A_1(n),$$

so the lemma is true.

Suppose the lemma holds for ordinals less than η. If η is a successor ordinal,

$$\begin{aligned} A_{\omega^\eta}(n) &= A_{\mathrm{pred}(\omega^\eta,n)}(n) \\ &\ge A_{\omega^{\mathrm{pred}(\eta)}+1}(n) \end{aligned}$$

because $\omega^\eta \xrightarrow{g} \omega^{\mathrm{pred}(\eta)} + 1$ under $g(\iota) = n$ and $A_\iota(n) \ge A_\tau(n)$ if $\iota \xrightarrow{g} \tau$ (see [10]),

$$= A^{(n)}_{\omega^{\mathrm{pred}(\eta)}}(1)$$

which, by induction,

$$\begin{aligned} &\ge A^{(n)}_{\mathrm{pred}(\eta)}(1) \\ &= A_\eta(n). \end{aligned}$$

If η is a limit ordinal,

$$\begin{aligned} A_{\omega^\eta}(n) &= A_{\omega^{\mathrm{pred}(\eta,n)}}(n) \\ &\ge A_{\mathrm{pred}(\eta,n)}(n) \end{aligned}$$

by induction, and

$$= A_\eta(n). \qquad \square$$

LEMMA 4.17. *For all ordinals* ι, $1 \le \iota < \epsilon_0$, *and for all integers* $n \ge 4$, $d_\iota(n) \le f_\iota(\alpha(n)) - 1$.

Proof. For $\iota < \omega$, $d_\iota(n) = 0$, so the result is trivial. Similarly, for $\iota = \omega$, $f_\omega(n) = n$ and hence

$$f_\omega(\alpha(n)) - 1 = \alpha(n) - 1 \ge 1 = d_\omega(n)$$

for $n \ge 4$. Thus the lemma is true for $\iota \le \omega$; we continue by transfinite induction. Suppose it is true for all ordinals less than ι, $\iota > \omega$. If ι is a successor ordinal then

$$\begin{aligned} d_\iota(n) &= d_{\text{pred}(\iota)}(n) \\ &\leq f_{\text{pred}(\iota)}(\alpha(n)) - 1 \end{aligned}$$

by induction;

$$< \quad f_\iota(\alpha(n)) - 1$$

by equation (3).

If $\iota < \epsilon_0$ is a limit ordinal, its fundamental sequence is one of two types. For $\iota = \omega^\eta \cdot \beta$, η and β successor ordinals, we have $\text{pred}(\iota, k) = \omega^\eta \cdot \text{pred}(\beta) + \omega^{\text{pred}(\eta)} \cdot k$. Thus,

$$\begin{aligned} d_\iota(n) &= d_{\text{pred}(\iota,\alpha_\iota(n))}(n) + 1 \\ &= d_{\omega^\eta \cdot \text{pred}(\beta) + \omega^{\text{pred}(\eta)} \cdot \alpha_\iota(n)}(n) + 1 \\ &\leq d_{\omega^\eta \cdot \text{pred}(\beta)}(n) + d_{\omega^{\text{pred}(\eta)} \cdot \alpha_\iota(n)}(n) + 1 \end{aligned}$$

by Lemma 4.15;

$$\leq \quad \left(f_{\omega^\eta \cdot \text{pred}(\beta)}(\alpha(n)) - 1\right) + \left(f_{\omega^{\text{pred}(\eta)} \cdot \alpha_\iota(n)}(\alpha(n)) - 1\right) + 1$$

by induction;

$$\begin{aligned} &= f_{\omega^\eta \cdot \text{pred}(\beta)}(\alpha(n)) + f_{\omega^{\text{pred}(\eta)} \cdot \alpha_\iota(n)}(\alpha(n)) - 1 \\ &= f_{\omega^\eta \cdot \text{pred}(\beta)}(\alpha(n)) + \alpha_\iota(n) f_{\omega^{\text{pred}(\eta)}}(\alpha(n)) - 1 \end{aligned}$$

by the definition of f_ι.

$$\leq \quad f_{\omega^\eta \cdot \text{pred}(\beta)}(\alpha(n)) + \alpha(n) f_{\omega^{\text{pred}(\eta)}}(\alpha(n)) - 1$$

because $\iota \xrightarrow{\forall} \omega$, so that $\alpha_\iota(n) \leq \alpha(n)$; and

$$= \quad f_\iota(\alpha(n)) - 1.$$

On the other hand, if ι is a limit ordinal of the form $\iota = \omega^\eta \cdot \beta$ where η is a limit ordinal and β is a successor ordinal, then

$$\begin{aligned} d_\iota(n) &= d_{\text{pred}(\iota,\alpha_\iota(n))}(n) + 1 \\ &= d_{\omega^\eta \cdot \text{pred}(\beta) + \omega^{\text{pred}(\eta,\alpha_\iota(n))}}(n) + 1 \\ &\leq d_{\omega^\eta \cdot \text{pred}(\beta)}(n) + d_{\omega^{\text{pred}(\eta,\alpha_\iota(n))}}(n) + 1 \end{aligned}$$

by Lemma 4.15,

$$\leq \quad d_{\omega^\eta \cdot \mathrm{pred}(\beta)}(n) + d_{\omega^{\mathrm{pred}(\eta,\alpha_\eta(n))}}(n) + 1;$$

by inequality (2) and Lemma 4.16.

$$\leq \quad \left(f_{\omega^\eta \cdot \mathrm{pred}(\beta)}(\alpha(n)) - 1\right) + \left(\alpha(n)^{f_{\mathrm{pred}(\eta,\alpha_\eta(n))}(\alpha(n))} - 1\right) + 1$$

by induction and the definition of f. However, by inequality (4), and since $\alpha(n) \geq \alpha_\eta(n)$,

$$\begin{aligned} &\leq \quad f_{\omega^\eta \cdot \mathrm{pred}(\beta)}(\alpha(n)) + \alpha(n)^{f_\eta(\alpha(n))} - 1 \\ &= \quad f_\iota(\alpha(n)) - 1. \end{aligned}$$

□

COROLLARY 4.18. *For all ordinals $\iota < \epsilon_0$ and for all integers $i \geq 0$, $d_\iota(n)$ is majorized by $\alpha_i(n)$.*

Proof. For $\iota < \epsilon_0$, $f_\iota(n)$ is clearly elementary, that is, in the Grzegorczyk class $\mathcal{E}^3$ ([7]; see [14] or [16], for example). Thus $f_\iota(n)$ is majorized by $A_i(n)$ for all integers $i \geq 2$, since these $A_i(n)$ are outside of $\mathcal{E}^3$. This means that $f_\iota(\alpha(n))$ is majorized by $A_i(\alpha(n))$ for all integers $i \geq 2$. However, $\alpha_i^{(2)}(n)$ majorizes $\alpha(n)$; applying A_i to each of these functions, we conclude that $\alpha_i(n) \geq A_i(\alpha_i^{(2)}(n))$ which majorizes $A_i(\alpha(n))$ and hence also majorizes $f_\iota(\alpha(n))$ and therefore $d_\iota(n)$. □

We are now ready to attack $L_\iota(n)$.

LEMMA 4.19. *For all ordinals $\iota < \epsilon_0$ and for all integers $n \geq 0$,*

$$L_\iota(n+1) \geq L_\iota(n) \tag{5}$$

and, if $\alpha_\iota(n+1) = \alpha_\iota(n) + 1$ and $\iota \geq 2$ then we have

$$L_\iota(n+1) \geq L_\iota(n) + \alpha_\iota(n) + 1. \tag{6}$$

Proof. For $\iota \leq 2$, (5) is obvious. For $\iota = 2$, $\alpha_\iota(n) = \lg^* n$; if $\lg^*(n+1) = (\lg^* n)+1$, then $\lfloor \lg(n+1) \rfloor \geq \lfloor \lg n \rfloor + 1$, $\lfloor \lg \lfloor \lg(n+1) \rfloor \rfloor \geq \lfloor \lg \lfloor \lg n \rfloor \rfloor + 1$, and so on—otherwise $\lg^*(n+1) = \lg^* n$, a contradiction. Each term in $L_2(n+1)$ is thus larger by 1 than the corresponding term in $L_2(n)$. Therefore,

$$\begin{aligned} L_2(n+1) - L_2(n) \quad &\geq \quad 1 + 1 + \cdots + 1 + L_1(\alpha_1^{(\alpha_2(n+1)-1)}(n+1)) \\ &= \quad \alpha_2(n) + 1, \end{aligned}$$

giving (6) for $\iota = 2$.

Continuing by transfinite induction, suppose (5) and (6) are true for all ordinals less than ι. There are two cases. First, if ι is a successor ordinal then by definition

$$\begin{aligned} L_\iota(n+1) \quad &= \quad L_{\mathrm{pred}(\iota)}(n+1) + L_\iota(\alpha_{\mathrm{pred}(\iota)}(n+1)) \\ &= \quad L_{\mathrm{pred}(\iota)}(n+1) + \sum_{i=1}^{\alpha_\iota(n+1)-1} L_{\mathrm{pred}(\iota)}(\alpha_{\mathrm{pred}(\iota)}^{(i)}(n+1)) \\ &\geq \quad L_{\mathrm{pred}(\iota)}(n) + \sum_{i=1}^{\alpha_\iota(n)-1} L_{\mathrm{pred}(\iota)}(\alpha_{\mathrm{pred}(\iota)}^{(i)}(n)) \end{aligned}$$

by induction, and

$$= L_\iota(n),$$

verifying (5). Furthermore, if $\alpha_\iota(n+1) = \alpha_\iota(n)+1$, then, as in the case $\iota = 2$, $\alpha^{(i)}_{\mathrm{pred}(\iota)}(n+1) = \alpha^{(i)}_{\mathrm{pred}(\iota)}(n)+1$ for $1 \le i \le \alpha_\iota(n)+1$, so by induction

$$\begin{aligned} L_\iota(n+1) &= L_{\mathrm{pred}(\iota)}(n+1) + \sum_{i=1}^{\alpha_\iota(n+1)-1} L_{\mathrm{pred}(\iota)}(\alpha^{(i)}_{\mathrm{pred}(\iota)}(n+1)) \\ &\ge L_{\mathrm{pred}(\iota)}(n) + \alpha_{\mathrm{pred}(\iota)}(n) + 1 \\ &\qquad + \sum_{i=1}^{\alpha_\iota(n)} \Big(L_{\mathrm{pred}(\iota)}(\alpha^{(i)}_{\mathrm{pred}(\iota)}(n)) + \alpha^{(i+1)}_{\mathrm{pred}(\iota)}(n) + 1 \Big) \\ &\ge L_{\mathrm{pred}(\iota)}(n) + 1 + \sum_{i=1}^{\alpha_\iota(n)} \Big(L_{\mathrm{pred}(\iota)}(\alpha^{(i)}_{\mathrm{pred}(\iota)}(n)) + 1 \Big) \\ &= L_\iota(n) + \alpha_\iota(n) + 1, \end{aligned}$$

verifying (6).

Second, if ι is a limit ordinal, $L_\iota(0) = L_\iota(1) = 0$, $L_\iota(2) = L_\iota(3) = 1$, and $L_\iota(4) \ge 1$, as is easily seen. Suppose $n \ge 4$. If $\alpha_\iota(n+1) = \alpha_\iota(n)$, then

$$L_\iota(n+1) = L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) - \alpha_\iota(n+1) + (0 \text{ or } 1)$$

and

$$L_\iota(n) = L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n) + (0 \text{ or } 1).$$

By induction,

$$L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) \ge L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n),$$

so (5) is proven if we can show

$$\alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) > \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n).$$

However, if these last two values are equal, by definition the "(0 or 1)" terms must be the same in both $L_\iota(n+1)$ and $L_\iota(n)$; this proves (5).

If $\alpha_\iota(n+1) = \alpha_\iota(n)+1$, for $n \ge 4$, we must prove that (6) holds. In this case,

$$L_\iota(n+1) = L_{\mathrm{pred}(\iota,\alpha_\iota(n)+1)}(n+1) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n)+1)}(n+1) - (\alpha_\iota(n)+1) + (0 \text{ or } 1)$$

and

$$L_\iota(n) = L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n).$$

(Examination of the definition of $L_\iota(n)$ shows that the "(0 or 1)" term must be zero in this last equation because $\alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) = \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(A_\iota(\alpha_\iota(n)+1)-1)$ and $A_\iota(\alpha_\iota(n)+1) = n+1$.) From inequality (1) we have

$$L_{\mathrm{pred}(\iota,\alpha_\iota(n)+1)}(n+1) \ge L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1),$$

and from Corollary 4.14 we have

$$\alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) = \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + 1.$$

Thus

$$\begin{aligned} L_\iota(n+1) &\geq L_{\mathrm{pred}(\iota,\alpha_\iota(n)+1)}(n+1) \\ &\geq L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n+1) \\ &\geq L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + 1 \end{aligned}$$

by induction;

$$\geq L_\iota(n) + \alpha_\iota(n) + 1,$$

proving (6). □

LEMMA 4.20. *Let $\sigma < \epsilon_0$ and let $k > 0$ be an integer. Then for all integers $n \geq 1$,*

$$L_{\sigma+k}(n) \leq L_\sigma(n) + (2^k - 1)\alpha_{\sigma+1}(n)^k L_\sigma(\alpha_\sigma(n)).$$

Proof. The proof is by induction on k. The lemma is trivial for $k = 0$. If $k \geq 1$,

$$\begin{aligned} L_{\sigma+k}(n) &= L_{\sigma+k-1}(n) + \sum_{i=1}^{\alpha_{\sigma+k}(n)-1} L_{\sigma+k-1}(\alpha^{(i)}_{\sigma+k-1}(n)) \\ &\leq L_{\sigma+k-1}(n) + \alpha_{\sigma+k}(n) L_{\sigma+k-1}(\alpha_{\sigma+k-1}(n)) \end{aligned}$$

by Lemma 4.19,

$$\begin{aligned} \leq\ & L_\sigma(n) + (2^{k-1} - 1)\alpha_{\sigma+1}(n)^{k-1} L_\sigma(\alpha_\sigma(n)) \\ & + \alpha_{\sigma+k}(n)\Big(L_\sigma(\alpha_{\sigma+k-1}(n)) \\ & \qquad + (2^{k-1} - 1)\alpha_{\sigma+1}(\alpha_{\sigma+k-1}(n))^{k-1} L_\sigma(\alpha_\sigma(\alpha_{\sigma+k-1}(n)))\Big), \end{aligned}$$

by induction. Taking $k = 1$ in much of the latter part of this last formula, the monotonicity of the various functions yields

$$\begin{aligned} \leq\ & L_\sigma(n) + (2^{k-1} - 1)\alpha_{\sigma+1}(n)^{k-1} L_\sigma(\alpha_\sigma(n)) \\ & + \alpha_{\sigma+1}(n)\left(L_\sigma(\alpha_\sigma(n)) + (2^{k-1} - 1)\alpha_{\sigma+1}(n)^{k-1} L_\sigma(\alpha_\sigma(n))\right). \\ \leq\ & L_\sigma(n) + (2^k - 1)\alpha_{\sigma+1}(n)^k L_\sigma(\alpha_\sigma(n)). \end{aligned}$$

□

COROLLARY 4.21. *Let $\sigma < \epsilon_0$ and let $k > 0$ be an integer. Then for all integers $n \geq 1$*

$$L_{\sigma+k}(n) \leq L_\sigma(n) + \left((2\alpha_{\sigma+1}(n))^k - 1\right) L_\sigma(\alpha_\sigma(n)).$$

LEMMA 4.22. *For any limit ordinal $\iota < \epsilon_0$, and for all $n \geq 1$,*

$$L_\iota(n) \leq L_2(n) + \Big((2\alpha_3(n))^{d_\iota(n)\alpha(n)} - 1\Big)\left(L_2(\alpha_2(n)) + \alpha_2(n)\right).$$

Proof. The proof is by transfinite induction. When $n \le 3$ the result is trivial for all ι, so assume $n \ge 4$. For the basis of the transfinite induction, let $\iota = \omega$; we have

$$\begin{aligned} L(n) &= L_{\alpha(n)}(n) + \alpha_{\alpha(n)}(n) - \alpha(n) + (0 \text{ or } 1) \\ &\le L_{\alpha(n)}(n) + \alpha_2(n). \end{aligned}$$

Applying Corollary 4.21 with $\sigma = 2$, $k = \alpha(n)$,

$$\le L_2(n) + \left((2\alpha_3(n))^{\alpha(n)} - 1\right)(L_2(\alpha_2(n)) + \alpha_2(n)),$$

so it is true for $\iota = \omega$.

Suppose the result holds for all limit ordinals less than $\iota > \omega$. By the definition of $L_\iota(n)$,

$$\begin{aligned} L_\iota(n) &= L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) - \alpha_\iota(n) + (0 \text{ or } 1) \\ &\le L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_2(n). \end{aligned}$$

Let $\mathrm{pred}(\iota, \alpha_\iota(n)) = \sigma + k$, where σ is a limit ordinal. If $k = 0$ we are done by induction; otherwise $0 < k = \alpha_\iota(n) \le \alpha(n)$. By Corollary 4.21 then,

$$\begin{aligned} L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) &= L_{\sigma+k}(n) \\ &\le L_\sigma(n) + \left((2\alpha_{\sigma+1}(n))^{\alpha(n)} - 1\right) L_\sigma(\alpha_\sigma(n)) \\ &\le L_\sigma(n) + \left((2\alpha_3(n))^{\alpha(n)} - 1\right) L_\sigma(\alpha_2(n)). \end{aligned}$$

Thus,

$$\begin{aligned} L_\iota(n) &\le L_{\mathrm{pred}(\iota,\alpha_\iota(n))}(n) + \alpha_2(n) \\ &\le L_\sigma(n) + \left((2\alpha_3(n))^{\alpha(n)} - 1\right) L_\sigma(\alpha_2(n)) + \alpha_2(n), \end{aligned}$$

and by induction,

$$\begin{aligned} \le\ & L_2(n) + \left((2\alpha_3(n))^{d_\sigma(n)\alpha(n)} - 1\right)(L_2(\alpha_2(n)) + \alpha_2(n)) \\ & + \left((2\alpha_3(n))^{\alpha(n)} - 1\right) \times \\ & \qquad \left(L_2(\alpha_2(n)) + \left((2\alpha_3(n))^{d_\sigma(\alpha_2(n))\alpha(n)} - 1\right)\left(L_2(\alpha_2^{(2)}(n)) + \alpha_2^{(2)}(n)\right)\right) \\ & + \alpha_2(n). \end{aligned}$$

However by an easy induction, $d_\sigma(n)$ is monotone in n, hence $d_\sigma(\alpha_2(n)) \le d_\sigma(n) = d_\iota(n) - 1$; so

$$\begin{aligned} \le\ & L_2(n) + (L_2(\alpha_2(n)) + \alpha_2(n)) \times \\ & \quad \Big((2\alpha_3(n))^{(d_\iota(n)-1)\alpha(n)} - 1 + (2\alpha_3(n))^{\alpha(n)} - 1 \\ & \qquad + ((2\alpha_3(n))^{\alpha(n)} - 1)((2\alpha_3(n))^{(d_\iota(n)-1)\alpha(n)} - 1)\Big) \\ \le\ & L_2(n) + (L_2(\alpha_2(n)) + \alpha_2(n))\left((2\alpha_3(n))^{d_\iota(n)\alpha(n)} - 1\right). \end{aligned}$$

□

THEOREM 4.23. *For any ordinal ι, $2 \leq \iota < \epsilon_0$,*

$$L_\iota(n) = L_2(n) + o(\lg\lg n).$$

Proof. Since $L_\iota(n) \geq L_2(n)$ by a straightforward transfinite induction, it is only necessary to prove that the right-hand side is an upper bound; this we do by transfinite induction. For $\iota = 2$ the result is obvious. If ι is a successor ordinal, then

$$L_\iota(n) \leq L_{\mathrm{pred}(\iota)}(n) + \alpha_\iota(n) L_{\mathrm{pred}(\iota)}(\alpha_{\mathrm{pred}(\iota)}(n)),$$

and by induction,

$$\begin{aligned} &= L_2(n) + o(\lg\lg n) + \alpha_\iota(n)(\lg \alpha_{\mathrm{pred}(\iota)}(n) + O(\lg\lg \alpha_{\mathrm{pred}(\iota)}(n))) \\ &= L_2(n) + o(\lg\lg n) + o((\lg^* n)(\lg\lg^* n)) + o((\lg^* n)(\lg\lg\lg^* n)) \\ &= L_2(n) + o(\lg\lg n). \end{aligned}$$

If ι is a limit ordinal, $d_\iota(n) = o(\lg^* n)$ from Corollary 4.18. Then from Lemma 4.22,

$$\begin{aligned} L_\iota(n) &\leq L_2(n) + (2\alpha_3(n))^{d_\iota(n)\alpha(n)}(L_2(\alpha_2(n)) + \alpha_2(n)) \\ &= L_2(n) + o\left((\lg^* n)^{(\lg^* n)^2}\right) O(\lg^* n) \\ &= L_2(n) + o(\lg\lg n). \end{aligned}$$

□

COROLLARY 4.24. *For any ordinal ι, $2 \leq \iota < \epsilon_0$,*

$$L_\iota(n) = \lg n + \Theta(\lg\lg n).$$

Theorem 4.23 can be improved by strengthening Lemma 4.22 to

$$L_\iota(n) \leq L_k(n) + \left((2\alpha_{k+1}(n))^{d_\iota(n)\alpha(n)} - 1\right)(L_k(\alpha_k(n)) + \alpha_k(n)),$$

for all integers $k \geq 2$. Applying this with $k+1$ in place of k we conclude

$$\begin{aligned} L_\iota(n) &\leq L_{k+1}(n) + o(\alpha_k(n)) \\ &\leq L_k(n) + \alpha_{k+1}(n) L_k(\alpha_k(n)) + o(\alpha_k(n)) \\ &\leq L_k(n) + O(\alpha_{k+1}(n) \lg \alpha_k(n)) + o(\alpha_k(n)) \\ &\leq L_k(n) + o(\alpha_k(n)). \end{aligned}$$

This improves considerably on the result in Corollary 4.24; for example, taking $k = 2$ yields

$$L_\iota(n) = L_2(n) + o(\lg^* n). \tag{7}$$

5. Conclusion. Clearly, generalized Ackermann's functions and corresponding infinite search algorithms can be obtained for ordinals beyond ϵ_0. However, our analysis of $L_\iota(n)$ depends critically on Cantor normal form, making it difficult to extend to $\iota \geq \epsilon_0$. We conjecture that (7) can be strengthened to state that if σ and τ are any ordinals, $2 \leq \sigma < \tau$, then

$$L_\tau(n) = L_\sigma(n) + O(\lg \alpha_\sigma(n)).$$

Acknowledgments. We gratefully acknowledge Richard Beigel for helpful comments and Micha Sharir, Nachum Dershowitz, and Carl Jockusch for pointing out various pertinent references.

REFERENCES

[1] W. Ackermann, *Zum Hilbertschen Aufbau der reellen Zahlen*, Math. Ann., 99 (1928), pp. 118–133.

[2] J. L. Bentley and A. C.-C. Yao, *An almost optimal algorithm for unbounded searching*, Inform. Process. Lett., 5 (1976), pp. 82–87.

[3] N. Dershowitz and E. M. Reingold, *Ordinal arithmetic with binary trees*, in preparation.

[4] P. Elias, *Universal codeword sets and representations of the integers*, IEEE Trans. Inform. Theory, 21 (1975), pp. 194–203.

[5] S. Feferman, *Systems of predicative analysis* II, J. Symbolic Logic, 33 (1968), pp. 193–220.

[6] R. L. Graham, B. L. Rothschild, and J. H. Spencer, *Ramsey Theory*, John Wiley & Sons, New York, 1980.

[7] A. Grzegorczyk, *Some classes of recursive functions*, Rozprawy Matematyczne, 4 (1953), pp. 3–45.

[8] P. R. Halmos, *Naive Set Theory*, Van Nostrand, Princeton, NJ, 1960.

[9] S. Kapoor and E. M. Reingold, *Optimum lopsided binary trees*, J. Assoc. Comput. Mach., 36 (1989), pp. 573–590.

[10] J. Ketonen and R. Solovay, *Rapidly growing Ramsey functions*, Ann. of Math., 113 (1981), pp. 267–314.

[11] D. E. Knuth, *Supernatural numbers*, in The Mathematical Gardner, D. A. Klarner, ed., Wadsworth International, Belmont, CA, 1981, pp. 310–325.

[12] M. H. Löb and S. S. Wainer, *Hierarchies of number-theoretic functions* II, Arch. Math. Logik, 13 (1971), pp. 97–113.

[13] M. Loebl and J. Nešetřil, *Linearity and unprovability of set union problems*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 360–366.

[14] R. Péter, *Recursive Functions*, 3rd ed., Academic Press, New York, 1967.

[15] E. M. Reingold and X. Shen, *More nearly optimal algorithms for unbounded searching, Part* I: *The finite case*, SIAM J. Comput., this issue, pp. 156–183.

[16] H. E. Rose, *Subrecursion: Functions and Hierarchies*, Oxford University Press, Oxford, 1984.

[17] D. Schmidt, *Built-up systems of fundamental sequences and hierarchies of number-theoretic functions*, Arch. Math. Logik, 18 (1976), pp. 47–53.

[18] C. Smoryński, *Some rapidly growing functions*, Math. Intelligencer, 2 (1980), pp. 149–154.

[19] S. S. Wainer, *Ordinal recursion, and a refinement of the extended Grzegorczyk hierarchy*, J. Symbolic Logic, 37 (1972), pp. 281–292.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 209-221, April 1991

MINIMUM WEIGHTED COLORING OF TRIANGULATED GRAPHS, WITH APPLICATION TO MAXIMUM WEIGHT VERTEX PACKING AND CLIQUE FINDING IN ARBITRARY GRAPHS*

EGON BALAS† AND JUE XUE†

Abstract. Efficient algorithms are known for finding a maximum weight stable set, a minimum weighted clique covering, and a maximum weight clique of a vertex-weighted triangulated graph. However, there is no comparably efficient algorithm in the literature for finding a minimum weighted vertex coloring of such a graph. This paper gives an $O(|V|^2)$ procedure for the problem (Algorithm 1). It then extends the procedure to the problem of finding in an arbitrary graph $G=(V, E)$ a maximal induced subgraph $G(W)$ color-equivalent (as defined in § 3) to a maximal triangulated subgraph $G(T)$ (Algorithm 2). Finally, it uses this latter algorithm as the main ingredient of a branch-and-bound procedure for the maximum weight clique problem in an arbitrary graph. Computational experience is presented on arbitrary random graphs with up to 2,000 vertices.

Key words. graph coloring, vertex packing, maximum clique finding, triangulated graphs

AMS(MOS) subject classifications. 05, 90, 68

1. Introduction.

Given an undirected graph $G=(V, E)$, a *stable set* (independent set, vertex packing) in G is a set of pairwise nonadjacent vertices, and a *clique* in G is a set of pairwise adjacent vertices. A clique in G is a stable set in $\bar{G}$, the complement graph of G, and vice versa.

A *vertex coloring* of G is an assignment of colors to the vertices of G in such a way that adjacent vertices get different colors. Equivalently, a vertex coloring is a collection of stable sets (color classes) such that each vertex belongs to at least one color class. A *clique covering* (of the vertices) of G is a collection of cliques such that every vertex belongs to at least one clique. A clique covering of G is a vertex coloring of $\bar{G}$, and vice versa.

A well-known pair of combinatorial optimization problems asks for finding a maximum clique and a minimum vertex coloring in G or, equivalently, for finding a maximum stable set and a minimum clique covering of $\bar{G}$. Both problems are NP-complete on general graphs, but solvable in $O(|E|+|V|)$ time on triangulated graphs, i.e., graphs that have no chordless cycles of length at least 4.

If the graph G has nonnegative integer weights on its vertices, say $w(v)$, $v \in V$, we have the weighted version of the above pair of problems. For the first problem, the weight of a clique is simply the sum of the weights of its vertices. On the other hand, for the second problem, a *weighted vertex coloring* $(y, \mathcal{S})$ of G is a collection $\mathcal{S} := \{S_1, \cdots, S_q\}$ of stable sets (color classes) along with nonnegative integer weights $y(S_i)$ of the color classes, such that for every vertex v, the sum of weights of the color classes containing v is at least equal to the weight of v. Thus a vertex can be assigned more than one color (can belong to more than one color class), but two adjacent vertices cannot be assigned the same color. A *minimum* weighted vertex coloring is one that minimizes the sum of weights of the color classes used.

* Received by the editors July 5, 1989; accepted for publication (in revised form) April 30, 1990. This research was supported by National Science Foundation grant ECS-8601660, Office of Naval Research contract N00014-85-K-0198, and Air Force Office of Scientific Research grant AFOSR-370292. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

† Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

Let $\mathcal{S}$ be the collection of all maximal stable sets (independent sets, vertex packings) of G; then the *maximum weight clique problem* on G is

$$\text{(WK)}\qquad \begin{aligned} &\max \sum (w(v)x(v)\colon v \in V) \\ &\text{s.t.} \\ &\sum (x(v)\colon v \in S) \leqq 1, \qquad S \in \mathcal{S} \\ &x(v) \text{ integer}, \quad x(v) \geqq 0, \qquad v \in V, \end{aligned}$$

while the *minimum weighted vertex coloring problem* is

$$\text{(WS)}\qquad \begin{aligned} &\min \sum (y(S)\colon S \in \mathcal{S}) \\ &\text{s.t.} \\ &\sum (y(S)\colon v \in S) \geqq w(v), \qquad v \in V \\ &y(S) \text{ integer}, \quad y(S) \geqq 0, \qquad S \in \mathcal{S}. \end{aligned}$$

It is well known that if G is perfect (i.e., for every induced subgraph of G, a maximum clique and a minimum coloring of the subgraph have the same cardinality), then the linear programming relaxations of (WK) and (WS), obtained by removing the integrality conditions from both constraint sets, have integer optimal solutions equal in value [4]. Furthermore, Grötschel, Lovász, and Schrijver [9] have shown that for perfect graphs both problems are solvable in polynomial time. Their approach uses the ellipsoid method for linear programming.

When G is triangulated (chordal) and unweighted, finding a maximum clique and a minimum coloring, as well as a maximum stable set and a minimum clique covering, can be done in $O(|E|+|V|)$ time, as shown by Gavril [7]. As to the weighted versions of these problems, since a triangulated graph G has at most $|V|$ maximal cliques, a maximum weight clique can be found by listing all maximal cliques and picking the one with maximum weight. A maximum weight stable set and a minimum weighted clique covering in a triangulated graph can be found in $O(|V|^2)$ time by a method due to Frank [6].

However, no special method has been published so far for solving the minimum weighted vertex coloring problem on a triangulated graph. In this paper we give such an algorithm, whose time complexity is $O(|V|^2)$. We then extend the algorithm to the problem of finding in an arbitrary graph $G=(V,E)$ a maximal induced subgraph $G(W)$ that is color-equivalent (as defined in § 3) to a maximal triangulated induced subgraph $G(T)$. Finally, we show how this latter algorithm can be used as the main ingredient of a branch-and-bound method for solving the maximum weight clique problem on an arbitrary graph, in a vein similar to that used in Balas and Yu [2].

2. Minimum weighted coloring of triangulated graphs. Given a graph $G=(V,E)$, and a set $S \subseteq V$, we write $G(S)$ for the subgraph of G induced by S. A vertex $v \in V$ is called *simplicial* if all vertices adjacent to v are pairwise adjacent to each other, i.e., if v is contained in a unique maximal clique. An ordering $\sigma=(v_1,\cdots,v_n)$ of the vertices of G is called a *perfect elimination scheme* if, for $i=1,\cdots,n$, v_i is simplicial in $G(\{v_i, v_{i+1},\cdots,v_n\})$. It is well known (see, for instance, Golumbic [8]) that a graph is triangulated if and only if it admits a perfect elimination scheme. Finding such a scheme or showing that it does not exist takes $O(|E|+|V|)$ time (see Rose, Tarjan, and Lueker [12]). We denote by $\sigma^{-1}(v_i)$ the rank (position) of v_i in σ. Further, we write $B(v)$ and $A(v)$ for the set of vertices adjacent to v that are before v and after v,

respectively, in σ; i.e., $B(v) := \{u \in V: (u, v) \in E, \ \sigma^{-1}(u) < \sigma^{-1}(v)\}$ and $A(v) := \{u \in V: (u, v) \in E, \ \sigma^{-1}(u) > \sigma^{-1}(v)\}$. We call $B(v)$ and $A(v)$ the set of predecessors and successors, respectively, of v, and consider both $B(v)$ and $A(v)$ ordered by σ. Finally, for any sequence σ, we write $\{\sigma\}$ for the (unordered) set of elements in σ. From the above property of triangulated graphs it follows that every maximal clique of G is of the form $\{v_i\} \cup A(v_i)$ for some $v_i \in \sigma$. Each maximal clique of G will also be considered ordered by σ.

Let $\sigma = (v_1, \cdots, v_n)$ be a perfect elimination scheme for the triangulated graph $G = (V, E)$, with $n = |V|$. Our algorithm consists of two steps applied iteratively: Step 1 determines the membership of a color class (stable set) S_i; Step 2 determines its weight $y(S_i)$ and updates the vertex weights $w(v)$ and the membership of the sequence σ of vertices not yet fully colored. (A vertex is said to be *fully colored* when its weight is matched by the sum of weights of the color classes to which it belongs.)

The algorithm uses two kinds of labels: when a vertex v is included into the color class S_t under construction, it gets the label 0 and all its predecessors get the label v. When a vertex w of the previous color class S_{t-1} is removed (not retained for S_t), its label 0 is erased and the labels w of its predecessors are also erased. Since the current color class S_t is constructed from the previous one (S_{t-1}), a vertex v whose successor in S_t is different from the one in S_{t-1} may for a while have two nonzero labels (from the point where the new successor is introduced, to the point where the old successor is removed); but when the new color class S_t is completed, every vertex has exactly one label.

ALGORITHM 1.
Initialization. Set $k := 1$, $\sigma_1 := \sigma$, $S_1 := \varnothing$, all $v_j \in \sigma_1$ unlabeled.
Iterative Step. Let $\sigma_k = (v_{j_1}, \cdots, v_{j_p})$. Set $i := p$ and go to 1.
1. (*Fill up a color class*).

If v_{j_i} is unlabeled, set $S_k := S_k \cup \{v_{j_i}\}$, label v_{j_i} with 0, and label all $v_j \in B(v_{j_i})$ with v_{j_i}.

If v_{j_i} is labeled with both 0 and some nonzero label, set $S_k := S_k \backslash \{v_{j_i}\}$, erase the label 0 of v_{j_i}, and erase the label v_{j_i} of all $v_j \in B(v_{j_i})$.

In all other cases continue.

Set $i := i - 1$

If $i \geqq 1$, go to 1; otherwise go to 2.

2. (*Determine weight and update*). Set

$$y(S_k) := \min\{w(v): v \in S_k\}$$

$$w(v) := \begin{cases} w(v) - y(S_k) & \text{if } v \text{ is labeled with } 0 \\ w(v) & \text{otherwise} \end{cases}$$

$$\sigma_{k+1} := \sigma_k \backslash \{v: w(v) = 0\}.$$

If $\{\sigma_{k+1}\} = \varnothing$, stop: $y(S_1), \cdots, y(S_k)$ is minimum weighted coloring of G.

Otherwise, for every v_{j_i} labeled with 0 and such that $w(v_{j_i}) = 0$, erase the label v_{j_i} of all $v_j \in B(v_{j_i})$; set $S_{k+1} := S_k$, $k := k+1$; and go to the Iterative Step. □

THEOREM 1. *Algorithm 1 finds a minimum weighted coloring of G in* $O(|V|^2)$ *time.*

Proof. Each set S_k generated in Step 1 is stable by construction and is hence a valid color class, and each vertex is fully colored before it is removed from σ_k in Step 2. Thus when the algorithm stops, the vector y is a weighted coloring of the vertices of G. To prove that $\sum (y(S_i): i = 1, \cdots, k)$ is minimum, we will show that there exists a clique K^* of G that is intersected by every one of the color classes $S_1, \cdots, S_k$. It

then follows that the weight $w(K^*)$ $(:=\sum(w(v)\colon v\in K^*))$ of the clique K^* is equal to the value $\sum(y(S_t)\colon t=1,\cdots,k)$ of the coloring found by Algorithm 1.

For $t=1,\cdots,k$, let σ_t be the subsequence of σ generated at iteration t of the algorithm, and let G_t be the subgraph of G induced by the vertices in σ_t. To prove the existence of K^* with the above-mentioned property, we first show that for $t=1,\cdots,k$, the color class S_t constructed at iteration t (i) intersects every maximal clique of G_t, and (ii) contains exactly one successor of every $v\in\{\sigma_t\}\backslash S_t$.

In Step 1, we examine from right to left every vertex v_{j_i} of σ_t and put it into S_t unless v_{j_i} or one of its successors is already in S_t. Since every maximal clique K of G_t is of the form $\{v_{j_i}\}\cup A(v_{j_i})$ for some $v_{j_i}\in\sigma_t$, it follows that the first (from the right) vertex of K that does not belong to any maximal clique already "represented" by a vertex in S_t is included in S_t. Thus the only way the maximal clique K could end up not being intersected by S_t is if every vertex of K belonged to some clique $K_i\neq K$ represented in S_t by some vertex $v_{j_l}\notin K$. But then the leftmost vertex v_{j_h} of K would belong to more than one maximal clique of $G_t(v_{j_h},\cdots,v_{j_p})$—a contradiction. This proves (i). To see (ii), notice that if S_t contains more than one successor of some $v_{j_h}\in\{\sigma_t\}\backslash S_t$, then those successors must be adjacent to each other (since v_{j_h} is simplicial in $G_t(v_{j_h},\cdots,v_{j_p})$)—a contradiction.

On the other hand, there exists at least one maximal clique of G, say K^*, that has a vertex in every G_t, $t=1,\cdots,k$. To see this, note that every vertex of G_k is also a vertex of $G_{k-1},G_{k-2},\cdots,G_1$. It then follows that K^* intersects every color class S_t, hence $w(K^*)\geqq\sum(y(S_t)\colon t=1,\cdots,k)$; and since $w(K^*)$ cannot exceed the value of a coloring, the inequality holds with equality and $\sum(y(S_t)\colon t=1,\cdots,k)$ is minimum.

Next we calculate the complexity of Algorithm 1. The Initialization Step is $O(|E|+|V|)\leqq O(|V^2|)$, namely, the time required for finding σ.

During Step 1, all vertices are scanned in turn from right to left, and possibly labeled with 0 or have their label of 0 removed. Once scanned, a vertex is no longer examined during the given iteration (of Step 1). Before being scanned, a vertex v may be examined, and possibly given a label or have a label removed, if one of its successors is included into S_t, or if one of its successors is removed from S_t. However, either of these events occurs at most once during an application of Step 1, since a vertex once included into S_t stays in for the rest of the iteration, a vertex once removed from S_t stays out for the rest of the iteration, and every vertex of $\{\sigma_t\}\backslash S_t$ has exactly one successor in S_t, for $t=1,\cdots,k$. To summarize, every vertex of G_t is examined at most three times during one application of Step 1, i.e., one such application takes $O(|V|)$ time, and during the whole algorithm Step 1 takes $O(|V|^2)$ time.

During Step 2, finding the minimum weight over S_t and updating the vertex weights takes $O(|V|)$ time, hence $O(|V|^2)$ time for the entire algorithm. On the other hand, erasing the labels of the predecessors of vertices removed from S_t as a result of getting fully colored, takes $\deg(v)$ operations for each such vertex v; and since each vertex gets fully colored only once during the whole algorithm, the complexity of this operation is $O(\sum(\deg(v)\colon v\in V))=O(|E|)\leqq O(|V|^2)$. Thus the complexity of Step 2 is also $O(|V|^2)$ for the whole algorithm. □

In formulating the pair of problems (WK), (WS) in § 1, we have assumed that the vertex weights $w(v)$ are integers. If the $w(v)$ are nonnegative reals, the maximum weight clique problem is still an integer program; but in the minimum weighted coloring problem it no longer makes sense to require the weights $y(S)$ of the color classes to be integers. Rather, it makes sense in this case to allow the $y(S)$ to be nonnegative reals, like the $w(v)$, i.e., to replace the integer program (WS) with the corresponding linear program, say, (WSL). It should be obvious that the above algorithm solves

(WSL) just as well as it solves (WS), i.e., if applied to a problem in which the vertex weights are real numbers rather than integers, the algorithm will find a minimum weighted coloring with nonnegative real weights of the color classes.

Algorithm 1 is illustrated on an example in § 4.

3. An extension. Given a maximal triangulated induced subgraph (MTIS) $G(T)$ of an arbitrary vertex-weighted graph $G=(V, E)$, and a set $W \subseteq V$, $W \supset T$, we will say that the induced subgraph $G(W)$ is *color-equivalent* to $G(T)$, if $G(W)$ has a minimum weighted coloring $(y, \mathcal{S})$ whose restriction to $G(T)$ is a minimum weighted coloring of $G(T)$ of the same weight as $(y, \mathcal{S})$. We say that $G(W)$ is a *maximal induced subgraph color-equivalent* to $G(T)$ (with respect to $(y, \mathcal{S})$) if no vertex of $V \setminus W$ can be fully colored by extending some color classes of $\mathcal{S}$. The motivation for this comes from the wish to extend the Balas–Yu procedure [2] for finding a maximum (unweighted) clique in an arbitrary graph to the weighted case. In that approach, a MTIS $G(T)$ is first found, along with a maximum clique K^* and a minimum coloring C of $G(T)$. Since $G(T)$ is triangulated, $|K^*|=|C|$. The coloring C is then extended to a maximal (with respect to C) induced subgraph $G(W)$ of G, by adding to the induced subgraph all of those vertices that fit into some color class of C (i.e., are not adjacent to any vertex in that color class). Since $G(W)$ is then guaranteed to have no larger clique than K^*, one can use a branching rule based on the principle that any clique of G greater than K^* must contain at least one vertex in $V \setminus W$.

Consider now the following extension of Algorithm 1. Let $G(T)$ be a MTIS of G. Balas and Yu [2] give an $O(|E|+|V|)$ algorithm for finding such a subgraph. Let σ be a perfect elimination scheme for $G(T)$, and let K^* be a maximum weight clique of $G(T)$. Further, let τ be any ordering of the vertices in $V \setminus T$.

In Algorithm 1, every vertex not in S_t has exactly one successor in S_t. Since the graph $G(W)$ which Algorithm 2 is designed to construct extends beyond $G(T)$, a vertex $v \in W \setminus S_t$ may have more than one successor in S_t (if $v \in W \setminus T$). Thus when we include some vertex v_{j_i} into S_t and label with v_{j_i} all its predecessors, some of the latter may end up with several nonzero labels. Conversely, when a vertex v_{j_i} is removed from S_t and the label v_{j_i} of each of its predecessors is erased, some of the predecessors may still not be eligible for inclusion into S_t, because of other nonzero labels (coming from other successors in S_t) they may have.

ALGORITHM 2.

Initialization. Let $k := 1$, $\sigma_1 := (\tau, \sigma)$, $S_1 := \emptyset$, all $v_j \in \sigma_1$ unlabeled.

Iterative Step. Same as in Algorithm 1, except for: (i) the fact that v_{j_i} may have several nonzero labels, and (ii) the stopping rule, which becomes:

If $\{\sigma_{k+1}\} \cap T = \emptyset$, set $S_i := S_i \setminus \{\sigma_{k+1}\}$, $i=1, \cdots, k$. If any $v \in \{\sigma_{k+1}\}$ can be fully colored by inclusion into some of the color classes $S_1, \cdots, S_k$, proceed with the inclusion; then stop: $y(S_1), \cdots, y(S_k)$ is a minimum weighted coloring of $G(W)$, where W is the set of fully colored vertices.

THEOREM 2. *Algorithm 2 finds a maximal induced subgraph $G(W)$ of G, color-equivalent to $G(T)$, and a minimum weighted coloring $y(S_1), \cdots, y(S_k)$ of $G(W)$, in $O(|E| \cdot |V|)$ time.*

Proof. By construction, each S_i, $i=1, \cdots, k$ is independent and the variables $y(S_1), \cdots, y(S_k)$ satisfy the coloring constraints for all $v \in W$ and for no $v \in V \setminus (W)$. Hence, $G(W)$ is a maximal induced subgraph for which $y(S_1), \cdots, y(S_k)$ is a weighted coloring. To see that this weighted coloring is minimum, define $y(S_i \cap T) := y(S_i)$, $i=1, \cdots, k$. Then by the same reasoning as in the proof of Theorem 1, $y(S_1 \cap T), \cdots, y(S_k \cap T)$ is a minimum weighted coloring for $G(T)$ (with some color

classes possibly repeated, i.e., $S_i \cap T = S_j \cap T$ not excluded for some $i \neq j$), and $\sum (y(S_i \cap T): i = 1, \cdots, k) = \sum (y(S_i): i = 1, \cdots, k)$. Thus $G(W)$ is color-equivalent to $G(T)$ and $y(S_1), \cdots, y(S_k)$ is a minimum weighted coloring for $G(W)$.

As to the complexity of the Algorithm, during Step 1 of each iteration a vertex v may be examined at most $\deg(v)$ times (once for each successor in S_{t-1} or S_t). Thus, the complexity of Step 1 is $O(\sum \deg(v): v \in V) = O(|E|)$ for one iteration, i.e., $O(|E| \cdot |V|)$ for the whole algorithm. The complexity of Step 2 is the same as in Algorithm 1. Finally, the complexity of the last step (checking for the maximality of W) is again $O(|E| \cdot |V|)$. □

The next section illustrates the algorithm on a numerical example.

4. An example. Consider the graph G shown in Fig. 1, with the vertex weights $w(v)$ in boxes. Applying to G the procedure of Balas and Yu [2], we find the maximal triangulated induced subgraph $G(T)$, where $T := \{1, 2, 3, 4, 6, 8\}$, and the perfect elimination scheme $\sigma := (6, 2, 3, 4, 8, 1)$. We first illustrate Algorithm 1 on the triangulated graph $G(T)$, whose edges are drawn in heavy lines.

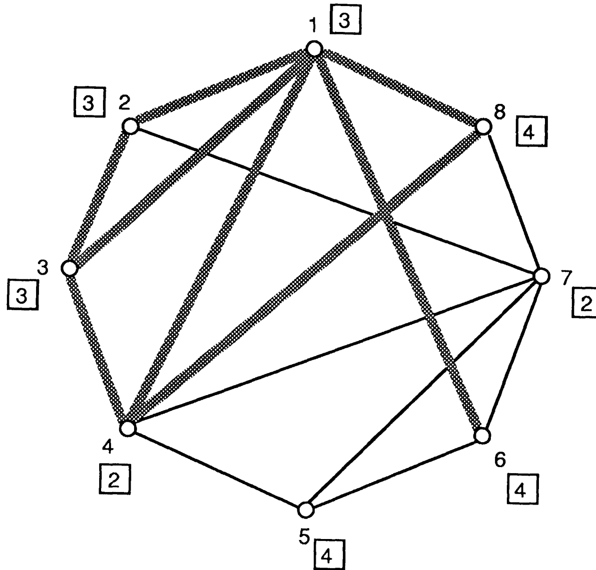


FIG. 1

At the start, $\sigma_1 := \sigma$ and the vertex weights for T, in the order of σ_1, are $w^1 := (4, 3, 3, 2, 4, 3)$. The algorithm puts vertex 1 into the color class S_1, labels it with 0, and labels with 1 the remaining vertices, as they are all predecessors of 1. It then assigns $y(S_1)$ the value 3, updates the vertex weights to be $w^2 = (4, 3, 3, 2, 4, 0)$, generates $\sigma_2 := \sigma_1 \backslash \{1\} = (6, 2, 3, 4, 8)$, and erases the labels of all predecessors of 1. Three more iterations follow until a minimum weighted coloring is obtained. A listing of the steps of the algorithm follows, with L representing the list of labels at the end of the step.

Initialization: $\sigma_1 := (6, 2, 3, 4, 8, 1)$, $w^1 := (4, 3, 3, 2, 4, 3)$,
$L := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

Iteration 1.

Step 1. $L := (1, 1, 1, 1, 1, 0)$, $S_1 := \{1\}$

Step 2. $y(S_1) = w(1) = 3$, $w^2 := (4, 3, 3, 2, 4, 0)$, $\sigma_2 := (6, 2, 3, 4, 8)$,
$L := (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

Iteration 2.

Step 1. $L := (0, 3, 0, 8, 0)$, $S_2 := \{8, 3, 6\}$

Step 2. $y(S_2) := w(3) = 3$, $w^3 := (1, 3, 0, 2, 1, 0)$, $\sigma_3 := (6, 2, 4, 8)$, $L := (0, \emptyset, 8, 0)$

Iteration 3.

Step 1. $L := (0, 0, 8, 0)$, $S_3 := \{8, 2, 6\}$

Step 2. $y(S_3) := w(6) = w(8) = 1$, $w^4 := (0, 2, 0, 2, 0, 0)$, $\sigma_4 := (2, 4)$, $L := (0, \emptyset)$

Iteration 4.

Step 1. $L := (0, 0)$, $S_4 := \{4, 2\}$

Step 2. $y(S_4) := w(2) = w(4) = 2$, $w^5 := (0, 0, 0, 0, 0, 0)$, $\sigma_5 := \emptyset$:

The total weight of the coloring is 9, which is also the weight of the two maximum weight cliques, $(1, 2, 3)$ and $(1, 4, 8)$.

To illustrate Algorithm 2, we apply it to G. For this purpose, we order the vertices of $V \setminus T$ into the (arbitrary) sequence $\tau = (5, 7)$, and obtain the sequence of steps listed below. Since vertices may now have multiple labels at the end of an iteration, labels of the same vertex will be separated by a comma, and labels of different vertices by a semicolon.

Initialization: $\sigma_1 := (5, 7, 6, 2, 3, 4, 8, 1)$, $w^1 := (4, 2, 4, 3, 3, 2, 4, 3)$, $L := (\emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset)$

Iteration 1.

Step 1. $L := (7; 0; 1; 1; 1; 1; 1; 0)$, $S_1 := \{1, 7\}$

Step 2. $y(S_1) := w(7) = 2$, $w^2 := (4, 0, 4, 3, 3, 2, 4, 1)$, $\sigma_2 := (5, 6, 2, 3, 4, 8, 1)$, $L := (\emptyset; 1; 1; 1; 1; 1; 0)$

Iteration 2.

Step 1. $L := (0; 1; 1; 1; 1; 1; 0)$, $S_2 := \{1, 5\}$

Step 2. $y(S_2) := w(1) = 1$, $w^3 := (3, 0, 4, 3, 3, 2, 4, 0)$, $\sigma_3 = (5, 6, 2, 3, 4, 8)$, $L := (0; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset)$

Iteration 3.

Step 1. $L := (6; 0; 3; 0; 8; 0)$, $S_3 := \{8, 3, 6\}$

Step 2. $y(S_3) := w(3) = 3$, $w^4 := (3, 0, 1, 3, 0, 2, 1, 0)$, $\sigma_4 := (5, 6, 2, 4, 8)$, $L := (6; 0; \emptyset; 8; 0)$

Iteration 4.

Step 1. $L := (6; 0; 0; 8; 0)$, $S_4 := \{8, 2, 6\}$

Step 2. $y(S_4)$: $w(6) = w(8) = 1$, $w^5 := (3, 0, 0, 2, 0, 2, 0, 0)$, $\sigma_5 := (5, 2, 4)$, $L := (\emptyset; 0; \emptyset)$

Iteration 5.

Step 1. $L := (4; 0; 0)$, $S_5 := \{4, 2\}$

Step 2. $y(S_5) := w(2) = w(4) = 2$, $w^6 := (3, 0, 0, 0, 0, 0, 0, 0)$, $\sigma_6 := (5)$, $L := (\emptyset)$; $\sigma_6 \cap T = \emptyset$.

At the last iteration, since $\sigma_6 \cap T = \emptyset$, we check whether any of the vertices left in σ_6 can be fully colored by extending the color classes at hand. Now $\{\sigma_6\} = \{5\}$. Of the color classes found, vertex 5 "fits" only into $\{1, 5\}$ and $\{4, 2, 5\}$. The weights of these classes are 1 and 2, respectively, while $w(5) = 4$. Thus 5 cannot be included in W. $G(W)$, with $W = T \cup \{7\}$, is a maximal induced subgraph of G, color-equivalent to $G(T)$; and $\{S_1, \cdots, S_5\}$ is a weighted coloring of $G(W)$ of the same total weight 9 as that found earlier for $G(T)$. Note that vertex 5, only partially colored, is not included in $G(W)$. □

5. Application to the maximum weight clique problem on an arbitrary graph. We now consider the maximum weight clique problem (WK) on an arbitrary graph $G=(V, E)$. As already mentioned, this is the same as the maximum weight vertex packing (stable set) problem on $\bar{G}$, the complement of G. We will extend to the weighted case the algorithm of Balas and Yu [2] for finding a maximum cardinality clique in a general graph. The basic idea of that approach is to perform branch-and-bound so as to generate only polynomially solvable subproblems.

As in the Balas-Yu algorithm, we start by finding a maximal triangulated induced subgraph (MTIS) $G(T)$ of G. This can be done by the $O(|E|+|V|)$ algorithm TRIANG described in [2], which requires only a trivial modification to also find a maximum weight clique in $G(T)$, say, K_0. Next we extend $G(T)$ to a maximal induced subgraph $G(W)$ for which K_0 is still a maximum weight clique. This is done by using Algorithm 2 of § 3 to find a maximal W, $T \subseteq W \subseteq V$, and a minimum weighted coloring $(y, \mathcal{S})$ of $G(W)$, such that $G(W)$ is color-equivalent to $G(T)$ (with respect to $(y, \mathcal{S})$). Then if $W=V$, we are done: K_0 is a maximum-weight clique of G. Otherwise we branch, based on the same considerations as in the unweighted case, stated in the next theorem. Let $N(v):=\{u \in V \backslash \{v\}: (u, v) \in E\}$.

THEOREM 3 (Balas and Yu [2]). *Let K_0 be a maximum weight clique of $G(W)$, and let $(v_1, \cdots, v_m)$ be an arbitrary ordering of the vertices in $V \backslash W$. If G has a clique K_1 such that $w(K_1) > w(K_0)$, then K_1 is contained in one of the m sets*

$$V_i := \{v_i\} \cup N(v_i) \backslash \{v_1, \cdots, v_{i-1}\}, \qquad i=1, \cdots, m,$$

where for $i=1$ we define $\{v_1, \cdots, v_{i-1}\} = \varnothing$.

We now describe the branching rule used in our algorithm. A node of the search tree is indexed by a string t and characterized by a pair $[I_t, E_t]$, where I_t and E_t are sets of vertices of G forcibly included in, and excluded from, the graph on which the current subproblem is defined. In the terminology of 0-1 programming, I_t and E_t are the index sets of variables fixed at 1 and 0, respectively, while the variables corresponding to $V_t := V \backslash (I_t \cup E_t)$ are free. Let $G(W_t)$ be a maximal induced subgraph of $G(V_t)$ for which K_t is a maximum weight clique, and let $V_t \backslash W_t = \{v_1, \cdots, v_m\}$. We generate m new nodes (subproblems) $t_1, \cdots, t_m$, by defining

$$(1) \qquad \begin{aligned} I_{ti} &:= I_t \cup \{v_i\} \\ E_{ti} &:= E_t \cup (V_t \backslash (\{v_i\} \cup N(v_i))) \cup \{v_1, \cdots, v_{i-1}\}, \qquad i=1, \cdots, m. \end{aligned}$$

The subproblem corresponding to node t of the search tree consists of finding a maximum weight clique in $G(V_t)$. (If K is a clique in $G(V_t)$, then $I_t \cup K$ is a clique in G.) A formal statement of the algorithm follows, with L, K^*, and P_t denoting the list of active nodes of the search tree, the clique with currently the largest weight, and the subproblem at node t (defined on the induced subgraph $G(V_t)$), respectively.

ALGORITHM 3.

0. *Initialize.* Put into L the problem defined on $G=(V, E)$. Set $t:=0$, $I_t = E_t := \varnothing$, $V_t := V$, $K^* := \varnothing$ and go to 1.

1. *Select a Subproblem.* If $L = \varnothing$, stop: the current clique K^* is of maximum total weight in G. Otherwise choose a subproblem P_t and remove it from L.

If $w(V_t \cup I_t) \leqq w(K^*)$, discard P_t and go to 1.

Otherwise go to 2.

2. Find a MTIS $G(T_t)$. Find a MTIS $G(T_t)$ of $G(V_t)$, along with a perfect elimination scheme σ_t and a maximum weight clique K_t of $G(T_t)$. If $w(I_t \cup K_t) > w(K^*)$, set $K^* := I_t \cup K_t$.

If $T_t = V_t$, discard P_t and go to 1. Otherwise go to 3.

3. *Find a maximal* $G(W_t)$ *color-equivalent to* $G(T_t)$. Apply Algorithm 2 to $G(V_t)$ with $\sigma := \sigma_t$ to find a maximal induced subgraph $G(W_t)$ of $G(V_t)$ color-equivalent to $G(T_t)$. If $W_t = V_t$, discard P_t and go to 1. Otherwise go to 4.

4. *Branch.* Order the vertex set $V_t \setminus W_t$ into a sequence $\{v_1, \cdots, v_m\}$. Generate m new subproblems defined by (1), place them into L, and go to 1.

6. Computational results. The above algorithms were implemented in C and tested on 114 random graphs of various densities (where density is the probability of a pair of vertices being joined by an edge, equal for all pairs), having between 100 and 2000 vertices and with costs randomly drawn from a uniform distribution of the integers between 1 and 10. The problems were run on a HP9000/835 workstation. Tables 1 and 2 summarize the results. There are two problems in each class, and the entries of the tables represent their averages.

The results are stated in the framework of the maximum weight clique problem on an arbitrary graph. This, of course, is the same as the maximum weight vertex packing problem on the complement graph.

As can be seen from the tables, problem difficulty for our algorithm increases with both the size and the density of the graph. This, of course, is not peculiar to our approach, but is intrinsic to the nature of the problem. To see why dense problems tend to be harder, it suffices to think of the fact that the size $k(G)$ of a maximum clique in G increases with the density of G, and the task of enumerating all the cliques is exponential in $k(G)$.

The most promising feature of Algorithm 3 is that, although the number of nodes of the search tree grows significantly with problem size and in particular with graph density, the algorithm invariably tends to find the maximum weight clique rather early in the procedure: at level 0–6 of the search tree for densities below 50 percent; at a somewhat higher level for higher densities (see the last column).

Until very recently, algorithms proposed for these problems were able to solve only instances of a much more modest size. Thus, the algorithms for weighted vertex packing discussed in Nemhauser and Trotter [10] and in Balas and Samuelsson [1] were tested on graphs with up to 50 vertices and 50 percent density. Recently, Pardalos and Rodgers [11] proposed a depth first branch-and-bound procedure for the (unweighted) maximum clique problem, based on an unconstrained quadratic 0–1 programming formulation, and Desai and Pardalos [5] implemented a version of the same approach addressing the maximum weight vertex packing problem. Subsequently, Carraghan and Pardalos [3] developed a simpler and apparently faster branch-and-bound procedure for the maximum clique problem. These codes are a substantial improvement over earlier attempts, in that they are able to solve much larger problems. Based on the detailed description of [5], we implemented the Desai–Pardalos procedure. Also, Panos Pardalos kindly made available to us the code of the weighted version of the Carraghan–Pardalos procedure whose unweighted version is described in [3]. We compared our algorithm with both of these procedures and the results are shown in Table 3. (The Desai–Pardalos algorithm, which is for the maximum weight vertex packing problem, was applied to the complement $\bar{G}$ of G to find a maximum weight clique in G.)

We will refer to the three algorithms by the initials of their authors. All three procedures use depth first branch-and-bound. The difference between DP and CP seems to be mainly in the search strategy (choice of subproblem to be developed next), and in computer implementation efficiency.

TABLE 1
Performance of Algorithm 3 on random graphs with 100–500 vertices.

$\|V\|$	%	Nodes of search tree	CPU seconds	Maximum clique weight	Weight of 1st clique found	Optimum found at level
100	10	13.0	0.06	28.5	28.5	0.0
100	20	32.0	0.09	37.0	35.5	0.5
100	30	51.0	0.15	45.5	45.5	0.0
100	40	72.0	0.19	57.0	53.0	2.0
100	50	99.5	0.27	69.5	62.0	4.0
100	60	198.5	0.55	84.0	83.0	2.5
100	70	301.5	0.97	104.5	101.0	1.5
100	80	387.0	1.71	137.0	135.0	4.0
100	90	269.5	1.84	213.5	208.5	14.0
200	10	68.0	0.26	34.5	34.5	0.0
200	20	113.0	0.47	49.0	43.0	1.0
200	30	244.0	0.83	56.5	49.5	3.0
200	40	890.0	2.45	68.5	63.0	2.0
200	50	1726.5	5.76	77.5	77.0	1.5
200	60	6411.0	23.21	96.5	93.5	6.0
200	70	18011.0	87.96	129.5	123.5	5.5
200	80	60038.0	536.52	186.5	184.5	12.0
300	10	152.5	0.75	36.5	34.5	2.0
300	20	258.0	1.29	46.5	45.0	2.5
300	30	1071.5	3.55	56.0	55.5	1.5
300	40	2961.0	10.51	73.0	70.5	1.5
300	50	13611.0	48.73	86.5	80.5	5.0
300	60	42078.0	208.66	117.5	103.5	5.5
300	70	427806.0	2590.35	150.0	141.0	8.5
400	10	250.5	1.48	35.5	35.5	0.0
400	20	539.0	2.94	50.0	49.0	1.5
400	30	3486.0	11.78	60.0	57.5	1.5
400	40	11379.0	43.08	78.5	76.0	2.0
400	50	58286.0	232.41	96.0	86.0	5.0
400	60	345979.0	1880.25	117.5	110.0	7.0
500	10	316.5	2.32	42.5	40.0	1.0
500	20	1510.5	6.19	53.0	47.5	3.5
500	30	6419.0	25.52	68.5	59.0	4.0
500	40	25902.5	120.25	83.5	75.0	3.5
500	50	179050.0	842.21	101.5	89.5	5.5

$|V|$: number of vertices.
%: density of the graph.

There are two main differences between these two procedures and the BX algorithm. First, while the DP and CP procedures use a straightforward heuristic to find a lower bound (a feasible solution), the BX algorithm uses, at certain nodes of the search tree, the more elaborate lower bounding procedure of finding a maximal triangulated induced subgraph $G(T)$ and a maximum clique of $G(T)$. Second, while none of the algorithms uses any upper bounding procedure in the usual sense of bounding from above all descendants of a given node, the BX procedure uses the minimum weighted coloring of $G(T)$ to find a maximal induced subgraph $G(W)$

TABLE 2
Performance of Algorithm 3 on random graphs with 600–2000 vertices.

$\lvert V\rvert$	%	Nodes of search tree	CPU seconds	Maximum clique weight	Weight of 1st clique found	Optimum found at level
600	10	499.0	3.56	39.0	39.0	0.0
600	20	3236.0	12.21	53.5	50.5	2.5
600	30	13247.5	53.87	67.5	64.5	2.5
600	40	58723.0	277.73	84.0	79.5	4.0
700	10	512.0	5.12	41.0	40.0	1.0
700	20	4541.0	20.41	58.0	56.0	1.0
700	30	20689.0	99.05	72.5	66.5	3.0
700	40	128439.0	660.08	88.0	80.5	4.5
800	10	683.0	7.20	43.5	41.0	1.0
800	20	7438.5	33.10	57.5	52.0	2.0
800	30	38414.5	191.33	71.0	69.0	4.0
800	40	305658.5	1429.39	89.0	84.5	5.5
900	10	712.0	10.58	46.0	45.5	1.0
900	20	12353.0	55.43	59.0	52.0	3.0
900	30	50771.0	297.69	73.5	68.0	4.0
1000	10	1017.5	13.25	47.0	41.5	2.0
1000	20	17581.5	82.72	58.5	53.0	3.5
1000	30	85078.5	503.91	75.5	70.5	3.5
1500	10	4043.0	43.42	47.5	45.0	2.0
1500	20	59633.5	373.13	66.0	60.0	3.5
2000	10	10928.5	106.47	49.0	49.0	0.0
2000	20	144626.0	1258.88	67.0	64.0	4.0

$\lvert V\rvert$: number of vertices.
%: density of the graph.

color-equivalent to $G(T)$, which provides an upper bound for all descendants of the current node associated with vertices in W and permits their immediate discarding.

As a result, the BX procedure can be expected to produce a much smaller search tree than the DP and CP algorithms, which is confirmed by the data of Table 3. At the same time, the BX algorithm spends more computational effort on every subproblem, and thus, for the comparison to be meaningful, it has to answer the question whether this extra effort is justified. The data on CPU time indicate that the BX algorithm is considerably faster than the DP procedure. It is of comparable speed with the CP algorithm on sparse and medium-density problems (up to 60 percent density), and substantially faster on high density problems (upwards of 70 percent), with the difference increasing fast with density.

In conclusion, it seems that the lower and upper bounds provided by the maximum weight clique of a triangulated subgraph and the minimum weighted coloring of a maximal induced color-equivalent subgraph can considerably speed up the performance of branch-and-bound algorithms for the maximum weight clique problem, especially for the most difficult class of problems, those on dense graphs. It should perhaps be mentioned that this class (maximum weight clique problems on dense

TABLE 3
Comparison of algorithms.

		Desai–Pardalos		Carraghan–Pardalos		Balas–Xue	
$\lvert V\rvert$	%	Nodes of search tree	CPU seconds	Nodes of search tree	CPU seconds	Nodes of search tree	CPU seconds
100	10	301.0	0.86	250.5	0.21	13.0	0.06
100	20	730.0	1.37	481.5	0.22	32.0	0.09
100	30	1200.0	2.225	1007.0	0.24	51.0	0.15
100	40	1893.5	3.1	1576.0	0.25	72.0	0.19
100	50	5477.0	8.22	3513.0	0.37	99.5	0.27
100	60	14286.5	21.49	11801.5	0.82	198.5	0.55
100	70	52014.5	80.79	36140.0	2.75	301.5	0.97
100	80	271583.0	469.555	202337.0	16.47	387.0	1.71
100	90	1424553.0	3000.02*	5336203.5	557.94	269.5	1.84
200	10	1549.5	6.355	834.5	1.03	68.0	0.26
200	20	3266.0	9.29	2135.5	1.08	113.0	0.47
200	30	9239.0	19.755	6250.0	1.28	244.0	0.83
200	40	41144.5	68.615	21536.5	2.01	890.0	2.45
200	50	129729.0	224.53	84271.5	5.31	1726.5	5.76
200	60	577034.0	1000.01*	428524.0	26.86	6411.0	23.21
200	70	1662354.0	3000.01*	2453445.5	176.15	18011.0	87.96
200	80	5075041.0	10000.01*	82429937.0	6628.08	60038.0	536.52
300	10	3736.0	21.055	2044.0	2.46	152.5	0.75
300	20	8898.5	33.425	7279.0	2.74	258.0	1.29
300	30	60344.0	118.965	26668.0	3.70	1071.5	3.55
300	40	174104.5	353.34	99465.0	7.91	2961.0	10.51
300	50	599293.0	1000.01*	593909.0	34.62	13611.0	48.73
300	60	530515.5	1000.01*	4355127.0	280.75	42078.0	208.66
300	70	5002772.0	9000.01*	88107834.0	6173.27	427806.0	2590.35

$\lvert V\rvert$: number of vertices.
%: density of the graph.
*: time limit exceeded without solution.

graphs, or maximum weight vertex packing on sparse graphs) is very frequent in applications.

REFERENCES

[1] E. BALAS AND H. SAMUELSSON, *A node covering algorithm*, Naval Res. Logis. Quart., 24 (1977), pp. 213–233.

[2] E. BALAS AND C. S. YU, *Finding a maximum clique in an arbitrary graph*, SIAM J. Comput., 15 (1983), pp. 1054–1068.

[3] R. CARRAGHAN AND P. PARDALOS, *An exact algorithm for the maximum clique problem*, Tech. Report, Computer Science Department, Pennsylvania State University, University Park, PA, 1990.

[4] V. CHVATAL, *On certain polytopes associated with graphs*, J. Combin. Theory Ser. B, 18 (1975), pp. 138–154.

[5] N. DESAI AND P. M. PARDALOS, *An algorithm for finding a maximum weighted independent set in an arbitrary graph*, Tech. Report, Computer Science Department, Pennsylvania State University, University Park, PA, 1989.

[6] A. FRANK, *Some polynomial algorithms for certain graphs and hypergraphs*, Proc. 5th British Combinatorial Conference, Winnipeg, Manitoba, Canada, 1975, pp. 211–226.

[7] F. GAVRIL, *Algorithms for minimum colorings, maximum clique, minimum covering by cliques, and maximum independent set of chordal graphs*, SIAM J. Comput., 1 (1972), pp. 180–187.

[8] M. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[9] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER, *Polynomial algorithms for perfect graphs*, Ann. Discrete Math., 21 (1989), pp. 325–356.

[10] G. L. NEMHAUSER AND L. E. TROTTER, JR., *Vertex packings: Structural properties and algorithms*, Math. Programming, 8 (1975), pp. 232–248.

[11] P. M. PARDALOS AND G. P. RODGERS, *A branch and bound algorithm for the maximum clique problem*, Tech. Report, Computer Science Department, Pennsylvania State University, University Park, PA, 1988.

[12] D. J. ROSE, R. E. TARJAN, AND G. S. LEUKER, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 222-227, April 1991

© 1991 Society for Industrial and Applied Mathematics

002

APPROXIMATE LEVELS IN LINE ARRANGEMENTS*

JIŘÍ MATOUŠEK†

Abstract. An arrangement L of n lines in the plane is considered. A *d-approximate level k* for L is a polygonal line meeting every vertical exactly once, such that for its every point p there are at least $k-d$ and at most $k+d$ lines of L above p. A *d-approximate leveling* for L is a collection $P_1, P_2, \cdots, P_{\lfloor n/2d \rfloor}$, where every P_i is a d-approximate level $2di$. A deterministic algorithm is given that, for a given L and a parameter $r \leqq n$, computes an (n/r)-approximate leveling for L, whose approximate levels consist of $O(r^2)$ segments in total.

The time complexity of this algorithm is $O(nr^{4/3}(\log n)^{4/3}(\log\log n + \log r)^c)$ (c a small constant), which is now outperformed by algorithms of Matoušek [*Proc. 5th Annual ACM Symposium on Computational Geometry*, 1989, pp. 1-10] and Agarwal [*Proc. 5th Annual ACM Symposium on Computational Geometry*, 1989, pp. 11-21]. A substantially different and elementary method is used here. An approximate leveling can be directly used for approximate halfplanar range counting, but it can also be used for plane partitioning applications, and thus as a subroutine in many recent computational geometry algorithms.

Key words. line arrangement, level, approximate level, deterministic algorithm, range counting, computational geometry

AMS(MOS) subject classification. 68Q25

1. Introduction, statement of results, and applications. We shall present a deterministic algorithm computing *approximate leveling* for an arrangement of lines in the plane. Let us introduce the necessary definitions first.

A finite set L of lines in the plane determines a cell complex in the plane, called the *arrangement* of L. We shall not distinguish between the arrangement and the underlying set of lines.

Let L be an arrangement of n nonvertical lines. The *level* k $(1 \leqq k \leqq n)$ of L is the set of points $X = (x_0, y_0)$, where y_0 is the maximum number such that the semiline $\{(x_0, y);\ y \geqq y_0\}$ meets at least k lines of L. It is easy to see that each level is an x-monotone polygonal line consisting of edges of the arrangement.

By a *layer* we shall mean a polygonal line meeting every vertical line exactly once. We say that a layer P is a *d-approximate level k* for L if it lies between levels $k-d$ and $k+d$ of the arrangement L (we take the closed region between the levels, and if $k-d<1$ then we only require P to lie above level $k+d$; similarly for the other end).

A *d-approximate leveling* for L is a sequence $P_1, P_2, \cdots, P_{\lfloor n/2d \rfloor}$, where P_i is a d-approximate level $2di$ for L. The *complexity* of an approximate leveling is the total number of segments of its polygonal lines. The notion of d-approximate leveling appeared in [Ma].

The paper of Edelsbrunner and Welzl [EW2] is perhaps the first paper which considers approximate levels and gives a method for their construction. This method first constructs the exact level and then its approximation. We shall consider this method in § 2.

A "global" construction of an approximate leveling was given in the original form of the present paper [Ma1] in 1987. Here we shall present the method of [Ma1], slightly improving the running time and the complexity of the leveling, and we shall prove the following theorem.

* Received by the editors October 19, 1987; accepted for publication (in revised form) June 13, 1990.

† Department of Computer Science, Charles University, Malostranské nám. 25, 118 00 Prague 1, Czechoslovakia. Present address, School of Mathematics, Georgia Institute of Technology, Atlanta, Georgia 30332.

MAIN THEOREM. Given an arrangement L of n lines and a number $r \leqq n$, one can find an (n/r)-approximate leveling of complexity $O(r^2)$ for L, deterministically and in time $O(nr^{4/3}(\log n)^{4/3}(\log\log n + \log r)^c)$, c a small constant.

(In order to avoid cumbersome calculations, we shall not try to optimize the value of c.)

Some of the ideas of [Ma1] were used in [Ma], where another deterministic algorithm (of time complexity $O(nr^2 \log r)$) for the same problem was given. This was further improved by Agarwal [Aga] to $O(n \log nr\ (\log r)^{\omega})$ ($\omega < 3.3$ is a constant). Let us also note that much simpler and even slightly more efficient methods are available if we permit randomized algorithms (see, e.g., [E&al] and [Aga]). However, the method used in [Ma], [Aga] was different from the one presented here, and quite heavy algorithmic tools were used (e.g., the AKS sorting network), while our algorithm will be elementary.

The immediate use of an approximate leveling is for an approximate halfplanar range counting, which has some significance since the known algorithms for an exact halfplanar range counting require either a large space or query time.

The *halfplanar range counting problem* is the following algorithmic problem: Given a set P of n points in the plane, preprocess it so that, given a query halfplane h, the number of points of P lying inside h can be determined quickly. The efficiency of query-answering in this problem depends on the amount of memory we are allowed to use (and also on the time we have for the preprocessing). A result of Chazelle [C] basically says that given $O(m)$ memory space, we can achieve a query time $O(n/\sqrt{m})$ at best. (Strictly speaking, this result does not quite apply to the halfplanar range counting problem, since the algorithm for which the lower bound works must be able to do something more general, but all known algorithms are of such a form.) After the effort of many authors (e.g., [EW1], [HW], [W]) and the combination of many previous results, a halfplanar range counting algorithm almost attaining the query time $O(n/\sqrt{m})$ with memory space m for a full range of m's (from $O(n \log n)$ to $O(n^2)$) was finally given in [Aga1].

However, in some applications we might be interested only in counting the number of points inside a query halfplane with a prescribed error only; this is what we call an *approximate halfplanar range counting*. By the well-known geometric duality (see, e.g., [E] for a definition), an equivalent problem is the following: Given a set L of lines, determine for a query point p the number of lines of L lying above p (maybe with some error in the approximate version).

Having an (n/r)-approximate leveling of complexity $O(r^2)$ for an arrangement L, we may preprocess the resulting subdivision of the plane for point location (e.g., by the algorithm of Kirkpatrick [K], in time and space $O(r^2)$). Then given a point p, we can find in time $O(\log r)$ the belt in our approximate leveling where p lies, and thus count the number of lines of L lying above p with error at most n/r. Thus the algorithm of the Main Theorem is a basis for a preprocessing step in an algorithm for the approximate halfplanar range counting.

Using such an algorithm, further approximate counting problems can be handled within similar time and space bounds (with some additional logarithmic factors). For example, applying the well-known range tree method, we can pass to approximate counting of points inside a query double wedge or inside a query triangle—see, e.g., [PS] for an explanation of the range tree method.

Perhaps a more significant application of an approximate leveling was found recently ([Ma], [Aga]). An (n/r)-approximate leveling of complexity $O(r^2)$ for an arrangement L of n lines can be straightforwardly used for partitioning the plane into

$O(r^2)$ (possibly unbounded) triangles, such that no triangle is intersected by more than $O(n/r)$ lines of L. Such a partitioning of the plane is a key to an efficient divide-and-conquer strategy in many planar geometric problems. We shall not give these applications here; they can be found in [Aga], [Aga1], [E&al], [CF].

In § 2, we give some more technical definitions. In §§ 3 and 4, we describe basic operations with approximate levelings. In § 5 we give the algorithm proving the main theorem, using the above operations.

2. Preliminaries. For the sake of simplicity we shall deal only with nondegenerate line arrangements. The results hold also for the general case, as one can show by a perturbation argument (simulation of simplicity, see [E]). We say that a set of lines L is *in general position* if no three lines of L meet at a common point, no two are parallel, and no two of their intersections have the same x-coordinate.

We shall need the following technical notion: A (d, A)-approximate leveling for an arrangement L of n lines is a collection $P_1, \cdots, P_{\lfloor n/d \rfloor}$, where the P_i is a dA-approximate level di (the number dA is called the *accuracy* of the approximate leveling). Note that a d-approximate leveling as defined in the previous section is a $(2d, \frac{1}{2})$-approximate leveling (*not* a $(d, 1)$-approximate leveling). Let us also remark that the notion of a (d, A)-approximate leveling is introduced just to optimize the time bound in the Main Theorem; with a slightly worse time bound, we could do with d-approximate levelings only.

In our algorithm, we shall need some "base case" method for the construction of a d-approximate leveling. The easiest such method is the following: Construct the whole arrangement (by [EOS], this can be done in time $O(n^2)$), and take the appropriate levels ($2di$, $i = 1, 2, \cdots, \lfloor n/d \rfloor$) as the d-approximate levels.

A more efficient method is to construct each of the levels $2id$ by the method of [EW2]. The construction of level k requires time $O(n \log n)$ plus $O((\log n)^2)$ per edge of the level, and from this we obtain the time bound for the construction of a d-approximate leveling of order $O(b(n, n/d) \cdot \log^2 n)$, where $b(n, r)$ denotes the maximum possible number of edges of r distinct levels in an arrangement of n lines. The best-known upper bound for $b(n, r)$ was given by Welzl [W1]; $b(n, r) = O(n^{3/2} r^{1/2})$. However, a famous conjecture about so-called k-sets (see [E] for references and discussion) essentially says that $b(n, 1) = O(n^{1+\delta})$ for every $\delta > 0$, and thus the validity of this conjecture would imply $b(n, r) = O(n^{1+\delta} r)$. Let us summarize our discussion in the following lemma.

LEMMA 1. *Given a collection L of n lines and a parameter $r \leqq n$, one can find an (n/r)-approximate leveling for L in time $O(b(n, r)\ (\log n)^2) = O(n^{3/2} (\log n)^2 r^{1/2})$. If the k-set conjecture holds, then the bound is $O(n^{1+\delta} r)$ for every $\delta > 0$.*

Finally, we shall quote two results of [Ma], the second one with an unessential modification, in the following two lemmas.

LEMMA 2. *For every arrangement of n lines and every $r \leqq n$, there exists an (n/r)-approximate leveling of complexity $O(r^2)$.*

LEMMA 3. *Let a and b be layers, consisting of n segments in total, such that a lies completely above b. Then a layer c lying between a and b and having a minimum number of segments among all such layers can be found in time $O(n)$.*

3. Simplifying an approximate leveling.

LEMMA 4. *Let given polygonal lines $P_1, P_2, \cdots, P_{\lfloor n/d \rfloor}$ form a (d, A)-approximate leveling of complexity M for an arrangement L of n lines. Then a $(2d, A+1)$-approximate leveling of complexity $O((n/d)^2)$ for L can be found in time $O(M)$.*

The proof of this lemma is a slight variation of the proof of Lemma 4.5 in [Ma]. The algorithm finding the new leveling is as follows (recall that P_i is an Ad-approximate level di):

For $i=1,2,\cdots,\lfloor n/2d\rfloor$, consider the region between approximate levels P_{2i-A-1} and P_{2i+A+1}. Using Lemma 3, find a layer R_i inside this region, which consists of as few segments as possible.

The first thing we have to show is that $R_1,\cdots,R_{\lfloor n/2d\rfloor}$ really form a $(2d, A+1)$-approximate leveling for L. Each R_i lies between P_{2i-A-1} and P_{2i+A+1}, and thus (by the properties of P_{2i-A-1} and P_{2i+A+1}) also between levels $(2i-A-1)d-Ad$ and $(2i+A+1)d+Ad$, and thus also between levels $2id-2(A+1)d$ and $2id+2(A+1)d$. This says precisely that R_i is a $2(A+1)d$-approximate level $2di$, as required.

We also have to show that this new approximate leveling has the claimed complexity. We note that the region between P_{2i-A-1} and P_{2i+A+1} must contain all the levels $2di-d$ through $2di+d$, thus also any d-approximate level $2di$, so that it suffices to know that for every arrangement of n lines there exists a d-approximate leveling of complexity $O((n/d)^2)$. However, this is what Lemma 2 says. □

4. Merging approximate levelings.

LEMMA 5. *Let $L_1, L_2,\cdots, L_k$ be line arrangements with m lines each and let a (d, A)-approximate leveling Λ_i of complexity at most C be given for each L_i. Then a (kd, A)-approximate leveling for $L=L_1\cup L_2\cup\cdots\cup L_k$ of complexity $M=O(k^2C(m/d))$ can be found in time $O(M\log M)$.*

Proof. Similarly as the levels for a line arrangement were defined, one can define the levels for an arrangement of polygonal lines. These levels can be computed by the technique of left-to-right plane sweeping in time $O(M\log M)$, where M is the number of edges plus the number of intersections of the polygonal lines (see, for example, [PS] for the details of the algorithm for segment intersection reporting by plane sweep).

Let us consider the arrangement Q of all polygonal lines of the Λ_i's. If x is a point of a level q in this arrangement of polygonal lines, this means that it has exactly q polygonal lines of Q above it, and thus there are at least $qd-kAd$ and no more than $qd+kAd$ lines of L above x. Therefore, the level q of the arrangement Q is an Adk-approximate level qd for L, and the levels ik $(i=1,2,\cdots,\lfloor m/d\rfloor)$ of Q form a (kd, A)-approximate leveling for L.

Now it remains to bound M, the total number of vertices of Q. The starting observation is the following: Given two x-monotone polygonal lines P and P', consisting of s and s' segments, then P and P' may intersect in at most $s+s'$ points (unless they have a common segment, which we may exclude by a perturbation argument).

Now let us consider two collections of polygonal lines, $P_1,\cdots,P_r$ and $P'_1,\cdots,P'_r$. If P_i has s_i segments and P'_i has s'_i segments, then the total number of intersections among the polygonal lines of the first collection and the polygonal lines of the second collection is at most

$$\sum_{i,j=1}^{r} s_i+s'_j=r\cdot(s_1+\cdots+s_r+s'_1+\cdots+s'_r).$$

Applying this on every pair of the k given approximate levelings, we get the bound $M=O(k^2C(m/d))$. This proves Lemma 5. □

5. Proof of Main Theorem. Let us describe the algorithm first, and then we determine the value of parameters occurring in its description and its time complexity.

Initial step. Divide the lines into groups by m_1 lines each. For every group first compute an $(m_1/r_0, 1)$-approximate leveling by the method of Lemma 1, and then, using Lemma 4, compute an $(m_1/r_1, 2)$-approximate leveling of complexity $O(r_1^2)$ for each group (it will be $r_1 = r_0/2$).

Step i $(i=1, 2, \cdots, I)$. At the beginning, we have $(m_i/r_i, i+1)$-approximate levelings of complexity $O(r_i^2)$ for the groups by m_i lines. We aggregate the groups into k_i-tuples, and we merge the groups in every k_i-tuple into a new group by $m_{i+1} = k_i m_i$ lines. By the procedure of Lemma 5 we compute an $(m_i k_i/r_i, i+1)$-approximate leveling for each new group from the approximate levelings for the old groups. Then, by the procedure of Lemma 4, we pass to $(m_{i+1}/r_{i+1}, i+2)$-approximate levelings of complexity $O(r_{i+1}^2)$ for the new groups.

After the Ith step, all the groups of lines will be merged into a single one, and we will have an $(n/r_{I+1}, I+2)$-approximate leveling of complexity $O(r_{I+1}^2)$ for our arrangement. We will arrange things so that $r_{I+1} \geq 4(I+2)r$ (recall that r is the number of approximate levels we want in the end). We then keep only every $(I+2)$nd approximate level, dropping the others, and we get an $(n/4r, 1)$-approximate leveling. We use the simplification step (Lemma 4) once more, yielding an $(n/2r, 2)$-approximate leveling of complexity $O(r^2)$, and this in turn gives us the desired (n/r)-approximate leveling of complexity $O(r^2)$ for the given arrangement.

We determine the parameters of the algorithm. In the sequel, the symbol "log" will mean logarithm base 2.

The value of m_1 will be determined in the end to balance the time complexity of the initial step and of the further steps. The other parameters will be described in terms of m_1.

The sequence $\{s_1, s_2, \cdots\}$, given by $s_1 = 2$, $s_i = 2^{2^{i-2}}$ $(i \geqq 2)$, satisfies the recurrence $s_{i+1} = s_i s_{i-1} \cdots s_1$ $(i \geqq 2)$. Let I be the smallest integer such that $m_1 s_{I+1} \geqq n$ (obviously $I \leqq \lceil \log\log n \rceil + 1$). We define $k_i = s_i$ for $i = 1, 2, \cdots, I-1$ and $k_I = n/(m_1 k_{I-1})$ and $m_i = k_{i-1} m_{i-1}$ for $i = 2, 3, \cdots, I$ (then we get $m_i = m_1 \cdot 2^{2^{i-2}}$ for $i = 2, 3, \cdots, I$ and $m_{I+1} = n$). Finally, we set $r_i = 4r(I+2)2^{I+1-i}$.

Let us analyze the time complexity of the algorithm. The initial step takes time $O(m_1^{3/2} r_1^{1/2} (\log m_1)^2)$ per group of lines (Lemma 1); thus $O(nm_1^{1/2} r_1^{1/2} (\log m_1)^2) = O(nm_1^{1/2}(r \log n \log\log n)^{1/2} (\log m_1)^2)$ in total.

The complexity of Step i for every newly formed group is by Lemmas 4 and 5 $O(k_i^2 r_i^2 \cdot r_i \log(k_i r_i))$, thus $O(n/(m_i k_i)\ k_i^2 r_i^3 \log(k_i r_i))$ in total, and substituting for k_i, r_i, and m_i, we get the bound:

$$O(n/m_1 \cdot (r \log n \log\log n \cdot 2^{-i+2})^3 \ (\log\log n + \log r + 2^{i-2}))$$

for the complexity of Step i.

Finally, the complexity of the last step will be of order r_{I+1}^2, which can be neglected compared to the other steps.

If we now put $m_1 = (r \log n)^{5/3}$, we get, after a routine calculation, that the complexity of the whole algorithm is $O(nr^{4/3}(\log n)^{4/3}(\log\log n + \log r)^c)$, as claimed. □

Let us remark that if the k-set conjecture were true (see § 2), we would get a better complexity of the initial step of our algorithm, $O(nm_1^{\delta} r_1)$ for every $\delta > 0$, and choosing $m_1 = (r \log n)^2$, the total complexity would be $O(n(\log n)^{1+\delta} r^{1+\delta})$. On the other hand, we might avoid the rather complicated algorithm of [EW2] for level construction (see Lemma 1) and construct the whole arrangement for the initial groups of lines instead; in this case, the total complexity comes out as $O(n(\log n)^{3/2} r^{3/2} (\log\log n + \log r)^c)$.

REFERENCES

[Aga] P. K. AGARWAL, *A deterministic algorithm for partitioning arrangements of lines and its applications*, Proc. 5th Annual ACM Symposium on Computational Geometry, Saarbrücken, FRG, 1989, pp. 11-21.

[Aga1] ———, *Ray shooting and other applications of spanning trees with low stabbing number*, Proc. 5th Annual ACM Symposium on Computational Geometry, Saarbrücken, FRG, 1989, pp. 315-325.

[CF] B. CHAZELLE AND J. FRIEDMAN, *A deterministic view of random sampling and its use in geometry*, Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 539-549; also in Combinatorica, 10 (1990), to appear.

[C] B. CHAZELLE, *Lower bounds for polytope range searching*, J. Amer. Math. Soc., 2 (1989), pp. 637-666.

[E] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, New York, 1987.

[E&al] H. EDELSBRUNNER, L. GUIBAS, J. HERSCHBERGER, R. SEIDEL, M. SHARIR, J. SNOEYINK, AND E. WELZL, *Implicitly representing arrangements of lines or segments*, Discrete & Comput. Geom., 4 (1989), pp. 433-466.

[EOS] H. EDELSBRUNNER, J. O'ROURKE, AND R. SEIDEL, *Constructing arrangements of hyperplanes with applications*, SIAM J. Comput., 15 (1986), pp. 341-363.

[EW1] H. EDELSBRUNNER AND E. WELZL, *Halfplanar range search in linear space and $O(n^{0.695})$ query time*, Inform. Process Lett., 23 (1986), pp. 289-293.

[EW2] ———, *Constructing belts in 2-dimensional arrangements*, SIAM J. Comput., 15 (1986), pp. 271-284.

[HW] D. HAUSSLER AND E. WELZL, *ε-nets and simplex range queries*, Discrete & Comput. Geom., 2 (1987), pp. 127-151.

[K] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28-35.

[Ma] J. MATOUŠEK, *Construction of ε-nets*, Proc. 5th Annual ACM Symposium on Computational Geometry, Saarbrücken, FRG, 1989, pp. 1-10.

[Ma1] ———, *Approximate halfplanar range counting*, KAM Series 59-87, Charles University, Prague, Czechoslovakia, 1987.

[PS] F. P. PREPARATA AND I. M. SHAMOS, *Computational geometry—an introduction*, Springer-Verlag, Berlin, New York, 1985.

[W] E. WELZL, *Partition trees for triangle counting and other range searching problems*, Proc. 4th Annual ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1988, pp. 23-33.

[W1] ———, *More on k-sets of finite sets in the plane*, Discrete & Comput. Geom., 1 (1986), pp. 95-100.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 228–245, April 1991
© 1991 Society for Industrial and Applied Mathematics
003

PARALLEL ALGORITHMS FOR CHANNEL ROUTING IN THE KNOCK-KNEE MODEL*

JOSEPH JÁJÁ† AND SHING-CHONG CHANG‡

Abstract. The channel routing problem of a set of two-terminal nets in the knock-knee model is considered. A new approach to route all the nets within d tracks, where d is the density, such that the corresponding layout can be realized with three layers is developed. The routing and the layer assignment algorithms run in $O(\log n)$ time with $n/\log n$ processors on the CREW PRAM model under the reasonable assumption that all terminals lie in the range $[1, N]$, where $N = O(n)$.

Key words. channel routing, parallel algorithms, layout, VLSI design, line packing, left-edge algorithm

AMS(MOS) subject classifications. 68C05, 68C25, 68E10

1. Introduction.

The recent advances in the VLSI technology allow the fabrication of highly complex systems on single chips. Sophisticated software tools are needed to successfully design such systems. In particular, the routing phase is a critical and time-consuming part of the overall design process. Unfortunately, it turns out that most routing problems are NP-complete, and hence no efficient algorithms to generate optimal solutions seem to be likely. There are few exceptions, however. For example, various river routing (one-layer) problems, the two-layer channel routing with no constraints, and few routing problems in the knock-knee model can be solved with efficient algorithms [Detal], [MP], [O], [P], [PL].

In this paper, we consider the channel routing problem of two-terminal nets in the knock-knee model. A routing algorithm that uses d tracks, where d is the density, is presented in [PL] such that the routing can be realized with three layers. This algorithm can be viewed as a nontrivial extension of the left edge algorithm [O] in which the routing is done row by row, left to right according to a greedy strategy. However, this method seems to be inherently sequential. We develop a new strategy to obtain an optimal routing (which is in general different from the one obtained by the [PL] method) such that both the routing and the layer assignment algorithms have linear time sequential implementations. Moreover, they both can be implemented on the CREW PRAM model in $O(\log n)$ time with $n/\log n$ processors, where n is the number of nets. Hence both algorithms are optimal in the sense that the corresponding total work is $O(n)$. In our analysis, we will make the reasonable assumption that all the terminals lie in the range $[1, N]$, where $N = O(n)$. If the terminals are arbitrary, our algorithms run in time $O(\log n)$ with n processors on the EREW PRAM model.

We use the PRAM (Parallel Random Access Machine) model where the available processors operate synchronously and have access to a shared memory unit. The EREW PRAM assumes that no two processors can read from or write into the same memory location, while the CREW PRAM allows concurrent read but no concurrent write. We will occasionally refer to the Common CRCW PRAM model, where concurrent read

* Received by the editors February 18, 1988; accepted for publication (in revised form) July 27, 1990.

† Department of Electrical Engineering, Institute for Advanced Computer Studies, Systems Research Center, University of Maryland, College Park, Maryland 20742. This work was supported in part by National Security Agency contract MDA-904-85H-0015, National Science Foundation grant DCR-86-00378, and by Systems Research Center contract OIR-85-00108.

‡ Department of Electrical Engineering, Systems Research Center, University of Maryland, College Park, Maryland 20742. Present address, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.

and concurrent write are allowed. For a concurrent write, the processors write the same value into a single memory location.

Parallel algorithms for computing prefix sums, list ranking, sorting, and computing all nearest smaller values are used frequently in our algorithms. Given n elements $a_0, a_1, \cdots, a_{n-1}$ in consecutive memory locations and an associative operator $*$, the prefix sums computation consists of evaluating the n partial sums $s_i = a_0 * a_1 * \cdots * a_i$, for $0 \leqq i \leqq n-1$. There is a simple optimal algorithm to solve the problem in time $O(\log n)$ with $n/\log n$ processors on the EREW PRAM model. If each a_i is a number that can be represented with $O(\log n)$ bits and $*$ is the addition operator, the prefix sums can be computed in time $O(\log n/\log\log n)$ time with $n \log\log n/\log n$ processors on the common CRCW PRAM model [CV]. If the a_i's belong to a linked list, then the corresponding problem can be handled optimally by using the list ranking algorithm [CV]. Arbitrary n numbers can be sorted in $O(\log n)$ time with n processors on the EREW PRAM model [C]. When the numbers involved are integers in the range $[1, N]$, where $N = O(n)$, each number occurring at most a constant number of times, and the identical numbers can be easily distinguished, the numbers can be sorted within the same time bound of computing prefix sums. In our case, the numbers are the terminals of a given set of nets and hence each number can occur at most twice, once as a top terminal and once as a bottom terminal. Such n numbers can then be sorted in $O(\log n)$ time with $n/\log n$ processors on the EREW PRAM, and in time $O(\log n/\log\log n)$ with $n \log\log n/\log n$ processors on the CRCW PRAM. The *all nearest smaller values* problem is defined in [BSV] as follows. Let $A = (a_1, a_2, \cdots, a_n)$ be n elements drawn from a totally ordered domain. For each a_i, find the nearest elements in A that are smaller than a_i, the left nearest smaller element a_j and the right nearest smaller element a_k with $j < i < k$ if they exist. This problem can be solved in $O(\log n)$ time using $n/\log n$ processors on the CREW PRAM and in time $O(\log\log n)$ time using $n/\log\log n$ processors on the CRCW PRAM [BSV]. For more details about shared memory algorithms, see [KR].

The rest of the paper is organized as follows. The basic definitions needed for the rest of the paper and our overall approach are introduced in the next section, while in § 3 we develop a routing strategy and establish its correctness. The layer assignment algorithm is presented in the last section.

2. Definitions. We assume that the reader is familiar with the basic definitions related to channel routing on a grid (see, for example, [O], [PL]). In this paper, we restrict ourselves to the two-terminal net channel routing problem (CRP) $\{N_i = \langle t_i, b_i \rangle \mid 1 \leqq i \leqq n\}$, where t_i is the *top* terminal on a gridpoint of the top boundary line and b_i is the *bottom* terminal on a gridpoint of the bottom boundary line. t_i and b_i will also represent the integer displacements of these terminals relative to a fixed origin. N_i is a *left* (*right*) net if $t_i < b_i$ ($t_i > b_i$). Otherwise it is a *vertical* net. We will also represent a net N_i as $N_i = [l_i, r_i]$, where $l_i \leqq r_i$, $l_i = \min\{t_i, b_i\}$ and $r_i = \max\{t_i, b_i\}$. We refer to l_i and r_i as the *left* and *right* terminals of N_i, respectively. The *local density* d_x at x is defined to be the number of nets $[l_i, r_i]$ such that $l_i \leqq x < r_i$. The *density* d is given by $d = \max_x \{d_x\}$. A *routing layout* in the knock-knee model consists of a set of edge-disjoint paths (made up of gridline segments) connecting the terminals of each net. Hence a shared gridpoint could be one of two types: crossing and knock-knee (Fig. 1(a)). For a layout with channel width t, we can use t horizontal lines (called *tracks*) to route all the nets. We assume that all the bottom (top) terminals are on track 0 (track $t+1$) and all the tracks in between are numbered 1 to t from bottom to

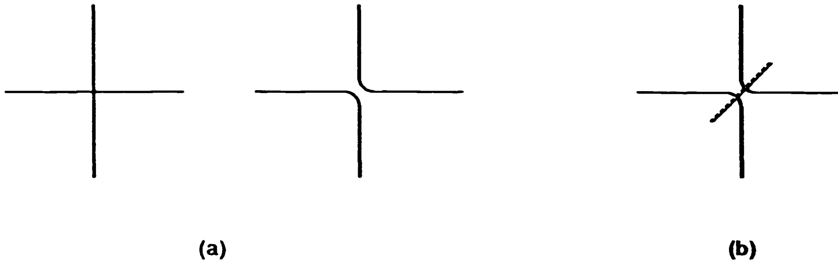


FIG. 1. (a) *Types of shared gridpoints*, (b) *diagonal representation of a knock-knee.*

top. A layout for a CRP is *optimal* if the channel width is minimum. Clearly, d is a lower bound on the channel width.

For any two terminals t_i and b_j of two different nets $N_i = \langle t_i, b_i \rangle$ and $N_j = \langle t_j, b_j \rangle$, a *vertical constraint* between t_i and b_j occurs if they are in the same column. If N_i is routed through a track at least as high as that of N_j, then the possible layouts are shown in Fig. 2(a). Otherwise we have to introduce a *detour*. Some possible detours are shown in Fig. 2(b). To introduce a detour for N_i, we have to find a *detour column* $dc(t_i)$ for terminal t_i as shown in Fig. 2(b) such that we can route terminal t_i to $dc(t_i)$ and then move to the track assigned to N_i. A *right detour* (*left detour*) is introduced when $dc(t_i)$ is to the right (left) of the terminal column. Figure 2(b)(1) is an example of a right detour and Fig. 2(b)(2) is an example of a left detour.

Let $L_1, L_2, \cdots, L_s$ be a set of conducting layers stacked on top of each other such that L_1 is on the bottom and L_t is on the top. A *wiring* of a layout is an assignment of single layer to each layout segment such that (1) no two segments of two distinct

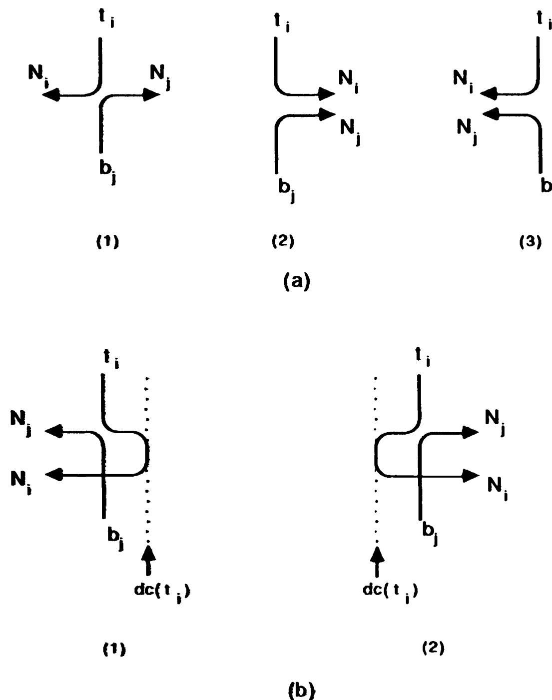


FIG. 2. *Some possible layouts of two nets with terminals in the same column.*

nets share a gridpoint on the same layer, (2) a routing path may change layers by a via at a gridpoint, and (3) no wire can use a gridpoint on a layer which is between two layers with a via at that gridpoint. It is known that any routing in the knock-knee model can be realized with four layers [BB] and that three layers suffice for the channel routing problem [PL].

A theory for wiring layouts has been developed in [LP], [PL], and [L]. One of the results shown is a characterization for layout wirability with three conducting layers. Since we will use this characterization to derive our layer assignment algorithm, we briefly introduce the necessary terminology. The reader is referred to these papers for a more formal presentation and for the proofs. Given a routing layout R of an instance of CRP, let D be the underlying grid determined by the channel and bounded at both ends by two vertical columns such that all the edges in R are contained in D (but not on the boundary of D). The *partition grid* $G(D)$ is the dual of D, i.e., the graph whose vertices are centers of the grid cells and whose edges join vertices belonging to adjacent cells (horizontal, vertical, and diagonal). Each knock-knee in R can be represented with an edge in $G(D)$ as shown in Fig. 1(b). The *diagonal diagram* of R is the set of edges of $G(D)$ representing all the knock-knees in R. A set P of edges of the partition grid $G(D)$ is called a *legal partition* if the following properties hold:

(1) Every internal vertex is incident on an even number of edges of P.
(2) The set of diagonals in P is identical to that of the diagonal diagram.
(3) None of the *forbidden patterns* in Fig. 3 appear in P.

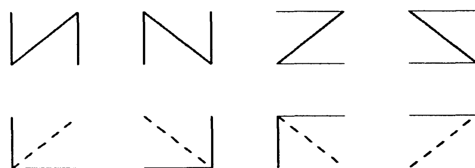


FIG. 3. *Forbidden patterns.*

A legal partition of a layout R exists if and only if R can be wired with three conducting layers.

Our basic approach is as follows. We first partition all the nets into d groups, where d is the density. Each group is defined as a chain (or list) of nets. If we view each net as an interval defined by its left and right terminals, then each chain consists of a set of nonoverlapping intervals. We then try to put each chain into a track to achieve an optimal routing. However, this initial partition may require complex detours to resolve vertical constraints and the final layout may not have an optimal three layer assignment. Hence we modify this partition by interchanging nets between chains to satisfy certain properties to be defined later. With this new partition, we assign one track to each chain and determine the layout very quickly. The corresponding diagonal diagram will have special properties which will be used to develop optimal algorithms for finding a legal partition and a three layer assignment after introducing some modifications to the diagonal diagram.

3. Channel routing. Given an instance of CRP of density d, our goal is to determine a layout of all the nets in d tracks. In addition, the resulting layout or a slight modification of it should be realizable in three layers.

The algorithm developed in [PL] constructs the layout track by track by laying each track from left to right. The overall strategy can be viewed as a nontrivial extension

of the *line packing* (or *left edge*) algorithm, where a mechanism is provided to solve conflicts arising in columns. Our method is different and consists of two main steps:

1. Partition the nets into d chains satisfying certain properties to be outlined below. In particular, the nets in each chain define a set of nonoverlapping intervals.
2. Assign a track number to each chain. Then wire all the nets simultaneously.

Our starting point is an initial partition of the nets into a set of d chains. We will denote the successor (predecessor) of a net N by succ (N) (pred (N)). The algorithm below appeared in [DS] and was independently discovered by the authors.

ALGORITHM CREATE CHAINS.
Input: terminals l_i's and r_i's of all nets $N_1, N_2, \cdots, N_n$.
Output: d chains of nets, where d is the density of the corresponding channel routing problem.

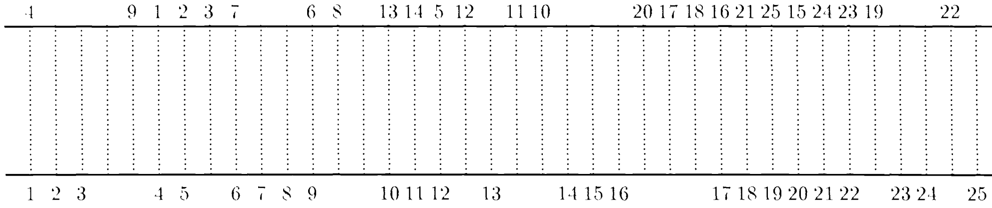
1. Sort all the terminals from left to right. If a right and a left terminal are equal, put the right terminal before the left. Otherwise, if two terminals are equal, put the bottom terminal before the upper terminal if both terminals are left. If both terminals are right, then the top terminal is placed before the bottom terminal.
2. Assign $+1$ to each left terminal and -1 to each right terminal. Compute the prefix sums of all the terminals.
3. For each right terminal r_i whose prefix sum is p, find the closest left terminal l_j to the right whose prefix sum is greater than p. Set succ $(N_i) = N_j$, if j exists. Otherwise set succ $(N_j) = \text{nil}$.

As an example, consider the channel routing instance of Fig. 4(a). The corresponding sorted list, prefix sums, and chains are shown in Figs. 4(b) and 4(c).

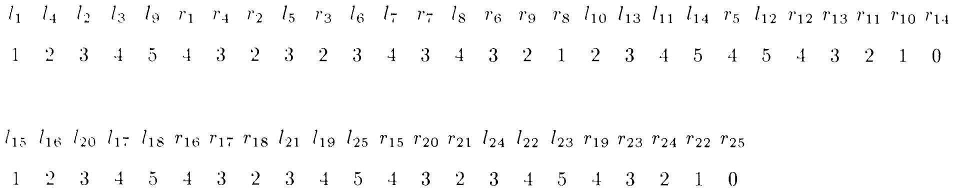
LEMMA 1. *The number of chains created by the algorithm above is exactly d, where d is the channel density. This algorithm can be implemented on the* CREW (CRCW, *respectively*) PRAM *in time* $O(\log n)$ $(O(\log n/\log\log n))$ *with* $n/\log n$ $(n \log\log n/\log n)$ *processors, where n is the number of nets.*

Proof. The correctness proof follows from [DS]. The running times follow from known algorithms for computing prefix sums and the nearest smaller values as indicated in the Introduction.

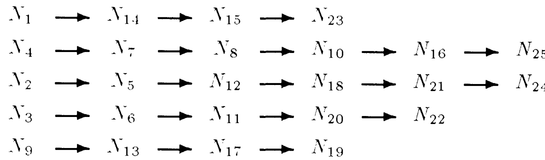
The chains above do not seem to be adequate for performing the routing fast. Complex detours may be required to resolve vertical constraints and the final layout may not possess a three-layer assignment. Consider, for example, the CRP given in Fig. 5(a). Algorithm Create Chains will generate the chain structure as shown in Fig. 5(b). Figure 5(c) shows a possible layout corresponding to this chain. To generate this layout in parallel, we have to identify all the detour columns very quickly. However, the detour column of terminal t may be in any column between t and the column of its successor (or predecessor) terminal t', and moreover, the location of this detour column depends on all the terminals between t and t'. Without additional constraints, it will be difficult to get all the detour columns in parallel. Even if we can find all the detour columns quickly, the corresponding diagonal diagram may be very complex. There may be an arbitrary number of diagonals in a column of the diagonal diagram. For example, there are five diagonals in the column of t_4 of Fig. 5(c). A complex diagonal diagram will complicate the task of finding a three layer assignment. To overcome these problems, we modify the chain structure such that we can determine the detour columns very quickly and the diagonal diagram can be simplified with at most two diagonals in each column.



(a)



(b)



(c)

FIG. 4. (a) *A* CRP, (b) *corresponding sorted list and prefix sums, and* (c) *chains.*

We modify the chains generated by *Algorithm Create Chains* so that they have the following property.

Column Property (CP): Let c be any column. Then either

1. c is empty, or
2. c contains one terminal, or
3. c contains two terminals t_i and b_j of nets N_i and N_j, respectively.
   - If one of b_j, t_i is a left terminal and the other is a right terminal, then both N_i and N_j belong to the same chain and one is the successor of the other.
   - Suppose that both b_j and t_i are right terminals. The other case can be dealt with similarly. Let $N_i' = \text{succ}(N_i)$ and $N_j' = \text{succ}(N_j)$. Then the left terminals of N_i' and N_j' either share a column or the column closer to c, say column c', has only one terminal. If they share a column, say $\hat{c}$, N_i' must be a right net and N_j' must be a left net. If the column c' has only a bottom (top) terminal, then the corresponding net must be the successor of N_i (N_j) (see Fig. 6(b)).

The following algorithm outlines how to modify the chains so that the above property holds.

ALGORITHM MODIFY CHAINS.

Input: A set of chains produced by the algorithm create chains.

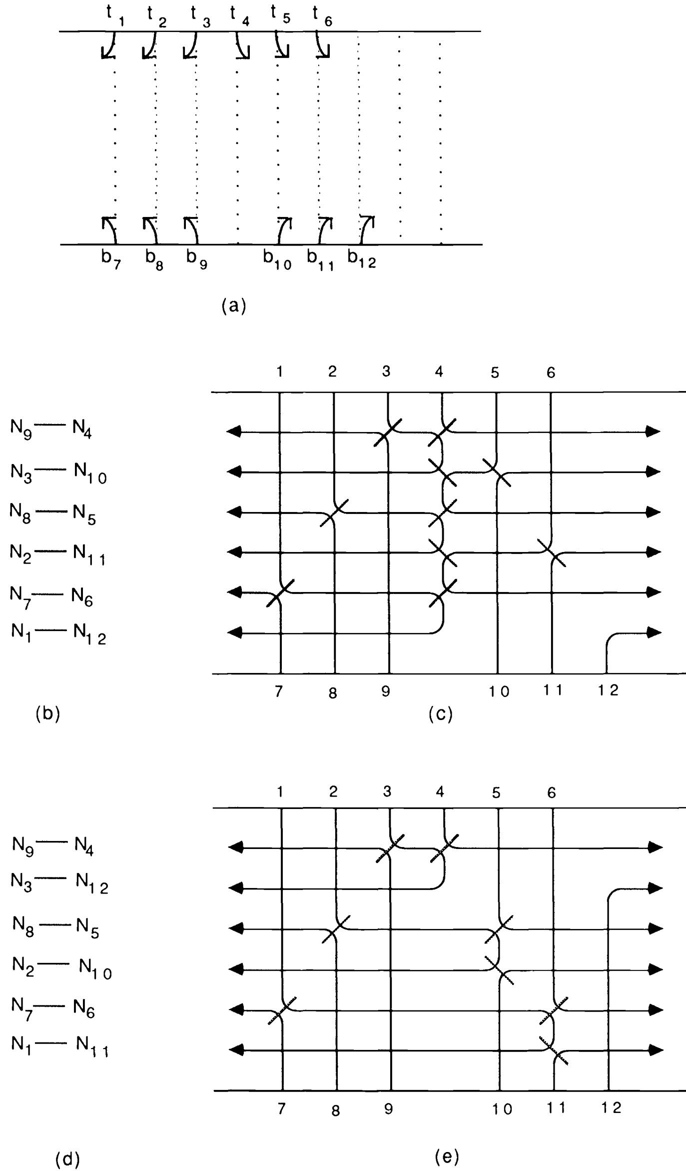


FIG. 5. *A* CRP *with layouts generated by Algorithms Create Chains and Modify Chains.*

Output: A set of chains satisfying the property (CP).

1. Mark each column with two right or two left terminals as active.
2. For each active column c with a top right terminal t_i and a bottom right terminal b_j, do the following:
   - If the left terminals of succ (N_i) and succ (N_j) are in the same column c', then mark both c and c' as inactive.
   - If the left terminals of succ (N_i) and succ (N_j) are in two distinct columns, c' and c'', say c' containing the left terminal of succ (N_j) is closer to c, then mark c inactive if c' has only one terminal. Otherwise, c' contains another left terminal b'_k. Let $N_k = \text{pred}\,(N'_k)$. Then create the pair $\langle N_i, N_k \rangle$. Mark c and c' as inactive.

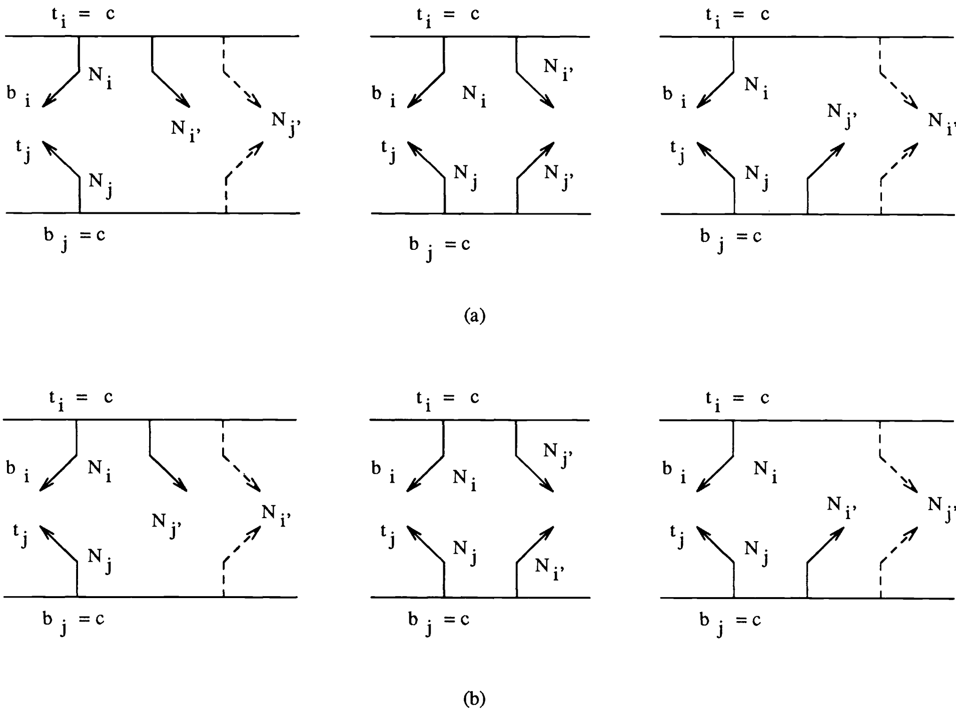


FIG. 6. *Possible successors of two nets with right terminals in the same column.*

3. Group the pairs $\langle N_i, N_k \rangle$ into maximal groups $\langle N_{k0}, N_{k1} \rangle$, $\langle N_{k1}, N_{k2} \rangle, \cdots,$ $\langle N_{kt-1}, N_{kt} \rangle$. Update the successors of these nets by setting the new successor of N_{ki} to be the previous successor of N_{ki+1} for all $0 \leqq i < t-1$. In addition, set the new successor of N_{kt} to be the previous successor of N_{k0}.
4. Repeat the procedure for active columns with two left terminals.
5. Adjust the chains in such a way that whenever the configurations of Fig. 6(a) occur, they will be replaced by the corresponding configurations of Fig. 6(b) (similarly for columns with two left terminals).

As an example, consider the chains of Fig. 5(b). In Step 2, when we detect the column of t_1 and b_7, the column of closer successor has another net N_{11} with predecessor N_2. Hence we construct the pair $\langle N_1, N_2 \rangle$. Similarly we will construct $\langle N_2, N_3 \rangle$. The final chain structure is shown in Fig. 5(d).

LEMMA 2. *The algorithm above modifies the chains generated by Algorithm Create Chains such that the new chains satisfy property* (CP). *Moreover, the algorithm runs in* $O(\log n)$ *time with* $n/\log n$ *processors on the* CREW PRAM *model.*

Proof. To simplify the presentation we will introduce a new graph called the *link graph.* There is a vertex v_c corresponding to each column c. There is an edge between v_c and $v_{c'}$ if and only if c contains a terminal of a net whose successor or predecessor has a terminal in c'. Let us call a column *good* if it satisfies property (CP) except possibly the ordering stated in Fig. 6(b).

Note that the link graph of each of the groups created in Step 3 has the form shown in Fig. 7(a). Let c_0' be the rightmost column involved in a group determined in Step 3. It is easy to check that if c_0' is connected to a, then a has only one terminal and appears as shown in Fig. 7(a). After the modifications of Step 3, the link graph

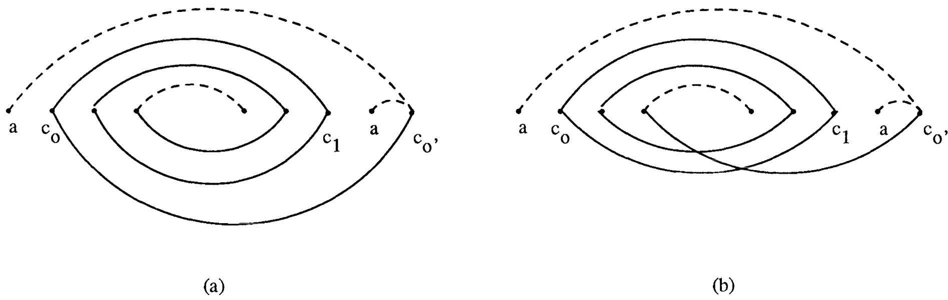


FIG. 7. *Forms of groups in the proof of Lemma* 2.

of the group will be of the form given in Fig. 7(b). Hence every column with two right terminals will be good after Step 3. Note that several columns with left terminals will be taken care of in this step and that every good column with left terminals will remain good after Step 3. However there might still be some columns with left terminals which are not good (e.g., column c_0'). Step 4 will take care of these columns.

As for the time complexity, Steps 1 and 2 are simple local operations, Step 3 can be easily done with list ranking. Similar comments apply to Steps 4 and 5. Actually the same time and processor bounds hold for the EREW PRAM model. Hence the lemma follows.

The track assignment and the wire layout will be described next. Suppose that track k has been assigned to net $N_i = \langle t_i, b_i \rangle$. Then the wire of N_i will consist of the interval $[l_i, r_i]$ on track k, a vertical line segment from b_i up to $[l_i, r_i]$, and a vertical line segment from t_i plus a possible detour to $[l_i, r_i]$. Therefore the problem comes down to determining how to connect a terminal on the upper row down vertically to its track. If t_i is in a column with only one terminal or there is another terminal in the same column which goes to a lower track than k, then we can connect t_i down to $[l_i, r_i]$ directly without any detour. Otherwise, the column of t_i has another bottom terminal b_j of a net N_j which is assigned to a higher track, say k'. Once property (CP) is satisfied, we can always find a left or right detour from t_i to $[l_i, r_i]$. If N_i is a right (left) net, then the detour column of t_i, $dc(t_i)$, must be in the column of the successor (predecessor) of N_i or N_j whichever is closer to t_i. It is easy to see that the line segments between t_i and $dc(t_i)$ on tracks k and k' are empty as well as the vertical line segment connecting tracks k and k' on column $dc(t_i)$. Hence it is easy to construct a right (left) detour using these three line segments. Each detour column can be determined in constant time with one processor once we have the modified chain structure. This seems unlikely for the chains generated by Algorithm Create Chain. Based on the above observations, we can assign an arbitrary track number to each chain and will always be able to generate a layout. But if we assign track numbers carefully, we can avoid the detours extending beyond the leftmost terminal. The algorithm below describes how to achieve this.

ALGORITHM WIRE NETS.
Input: A chain of nets as modified by Algorithm Modify Chains.
Output: A wire layout for each net.

1. For each chain, assign the leftmost terminal l_i as the primary key, and, if l_i is a bottom terminal, assign 0 as the secondary key, and 1 otherwise. Sort the chains according to their keys by using the bucket sort and the prefix sums algorithms. The track number of each chain is its corresponding rank.

2. For each column c, do the following:
   (1) If c contains one terminal of a net N, then connect that terminal vertically to the track of N.
   (2) Suppose c contains two terminals of a single net. Then connect these two terminals vertically.
   (3) Suppose that c contains two terminals of two distinct nets $N = \langle t_1, b_1 \rangle$ and $M = \langle t_2, b_2 \rangle$, where t_1 and b_2 lie on column c. If N and M have the same track number, then wire the terminals to this track using a knock-knee. Otherwise there is detour only if the track number of N is less than that of M. In this case, it is a left or right detour depending on whether c is a right or left terminal. The detour extends either to the column of successor (for a right detour) or predecessor (for a left detour) of either N or M whichever is closer. All the cases that can arise and the corresponding routing are shown in Fig. 8.

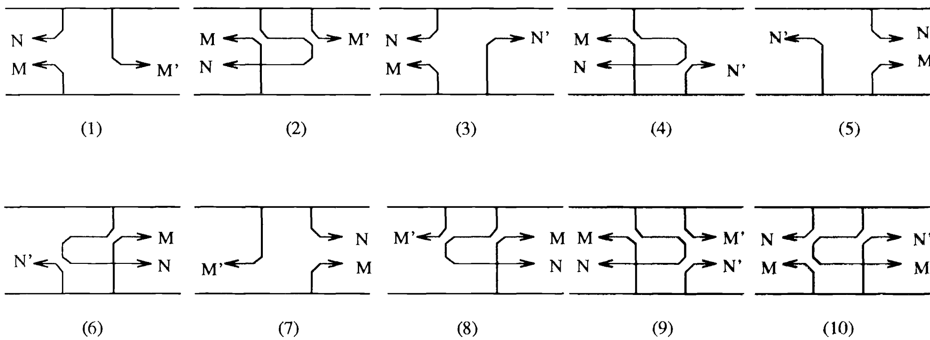


FIG. 8. *Possible detours of nets with terminals in the same column. N' and M' represent the successor or the predecessor of nets N and M with terminals sharing a column.*

Consider the example of Fig. 4 again. Then the routing obtained by the algorithm above is given in Fig. 9(a).

LEMMA 3. *Given an instance of the channel routing problem, the algorithm above provides a routing layout of all the nets in the knock-knee model. The time complexity of this algorithm is* $O(\log n)$ *with* $n/\log n$ *processors on the* CREW PRAM *model.*

THEOREM 1. *Given an instance of the channel routing problem of density d, it is possible to wire all the nets in d tracks in time* $O(\log n)$ *time on the* CREW PRAM *model with* $n/\log n$ *processors, where n is the number of nets.*

4. Layer assignment. In this section, we show that a modified version of the routing produced by the algorithm of the previous section can be laid out in three layers. Reference [PL] provides necessary and sufficient conditions for the realization of a wiring in three layers. As stated in § 2, the problem is essentially reduced to finding a legal partition. The routing layout produced by the algorithm in [PL] has a special property, namely, every column is either empty or contains one diagonal or a diagonal \ on the bottom and a diagonal / above it. Their algorithm proceeds from left to right, looking at each column and making vertical connections (and possibly changing the routing) so that the resulting partition is legal. Unfortunately, we encounter a major difficulty in our case. Each column of our routing layout could have two diagonals (\ and /) in an arbitrary order (because our routing uses left and right detours). This makes it necessary to change the wire layout more substantially than was done in [PL]. In the rest of this section, we outline how to overcome this difficulty.

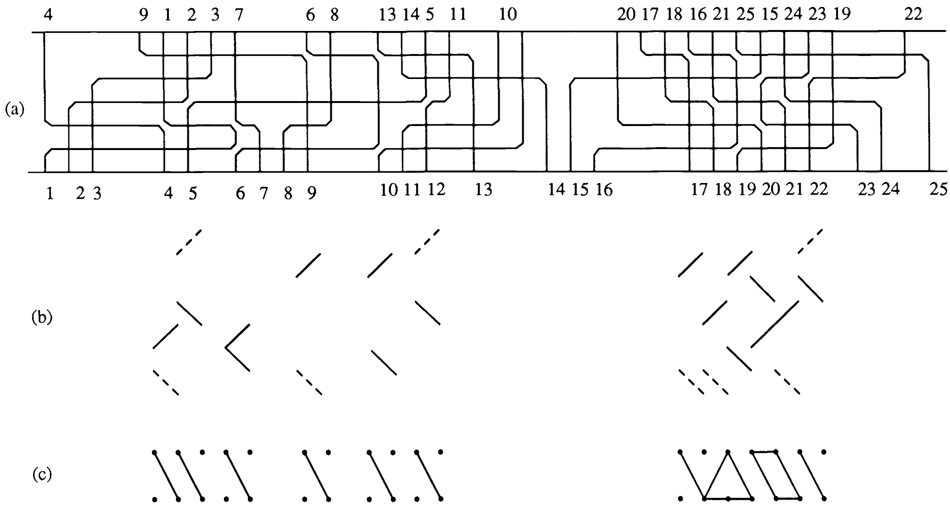


FIG. 9. (a) *The layout generated by Algorithm Wire Nets,* (b) *its corresponding diagonal diagram, and* (c) *its corresponding constraint graph.*

By adding dummy diagonals if necessary, we can assume that each column is either empty or contains exactly two diagonals. As in [PL], our partition will be constructed by adding vertical edges only. Define a *reference line* as a vertical line that touches the endpoint of some diagonal. For each reference line, the diagonals touching this line will partition it into several line segments. Number these line segments starting from the topmost segment. Note that there are two possible ways of adding vertical segments (to create a legal partition): add the odd-numbered or the even-numbered segments. We have to choose (if possible) those segments that will not create a forbidden pattern.

We define the *constraint graph* as follows. The two possible choices of vertical segments corresponding to reference line L_i are represented by two vertices v_{2i-1} and v_{2i}. Two vertices of the constraint graph are connected by an edge if and only if the corresponding choices create a forbidden pattern. Note that forbidden patterns can be created only between adjacent reference lines.

LEMMA 4. *The total number of edges between the vertices corresponding to adjacent reference lines is less than or equal to two.*

Proof. Since the maximum number of diagonals between two adjacent vertical reference lines is 2, there are at most two "constraints" between $\{v_{2i-1}, v_{2i}\}$ and $\{v_{2i+1}, v_{2i+2}\}$, for each i.

A *column* is a pair of vertices $\{v_{2i-1}, v_{2i}\}$ corresponding to a single reference line. Our goal is to select a vertex from each column in such a way that no two selected vertices are connected by an edge. A constraint graph will be called *legal* if such a selection is possible. The constraint graph corresponding to the layout produced by our algorithm may not be legal, in which case the routing layout has to be modified. Our strategy will be to identify the reference lines which might create problems (to be defined precisely), and modify the routing around these reference lines in such a way that the corresponding graph will become legal.

A simple case that can be handled immediately is when none of the vertices of the constraint graph has two edges adjacent to a single column. It can be easily verified that the constraint graph is legal in this case. A simple algorithm to select a proper subset of vertices is the following. Assign a weight of 0 to each reference line L_k if

there is an edge between v_{2k-3} and v_{2k} or between v_{2k-2} and v_{2k-1}. Otherwise assign a weight of 1. Compute the prefix sums of all reference lines and select v_{2k} if the rank of L_k is even; otherwise, select v_{2k-1}.

In the rest of this section, we will show how to modify the wiring in such a way that the corresponding constraint graph is legal. We first introduce the following classification of reference lines (cf. [PL]): *Trivial* (Fig. 10), *Overlap* (Fig. 11), *Disjoint* (Fig. 12), *Inclusion* (Fig. 13). Each type is shown with its possible constraint graph. Consider a reference line of type T_i. If one of the corresponding vertices has two edges adjacent to a single column, then we can select the vertex of degree 0 and remove this column (with the edges incident on it). Note that if the remaining graph is legal, then

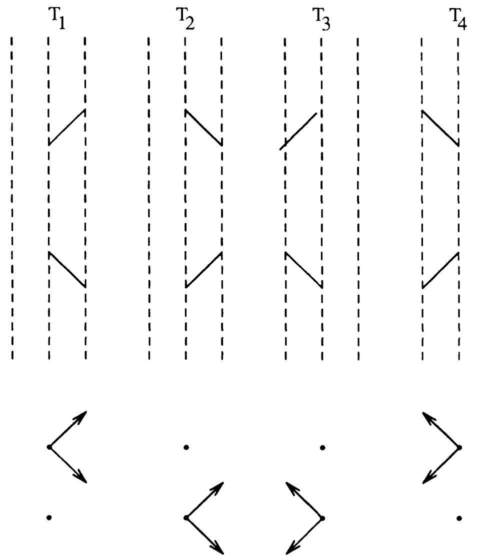


FIG. 10. *Trivial reference lines.*

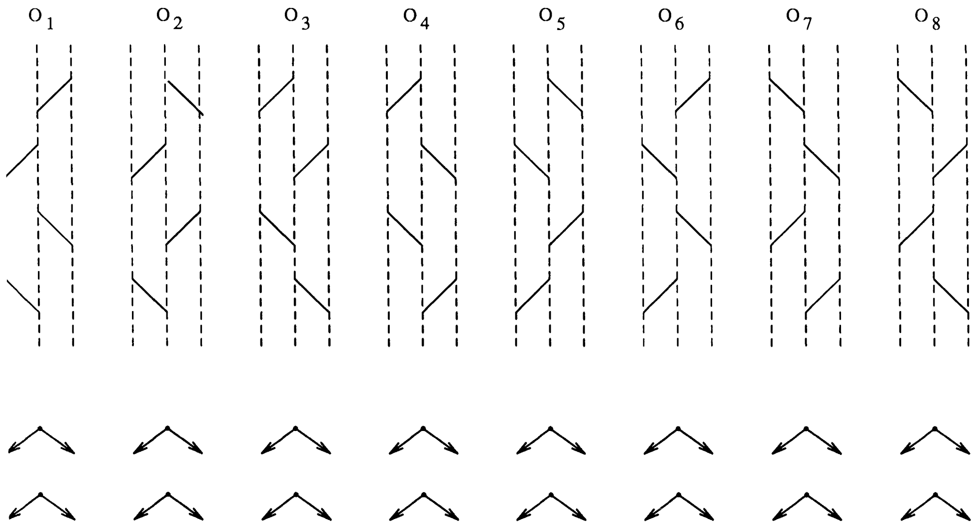


FIG. 11. *Overlap reference lines.*

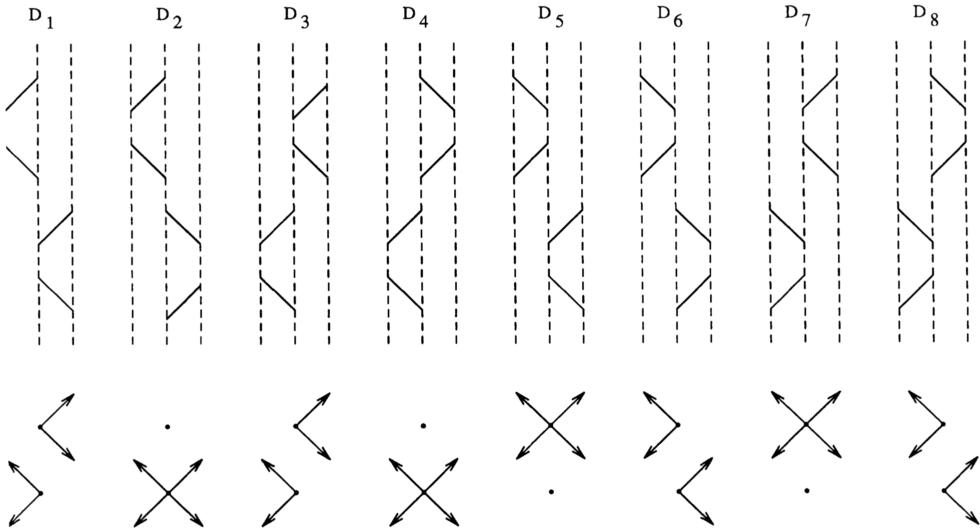


FIG. 12. *Disjoint reference lines.*

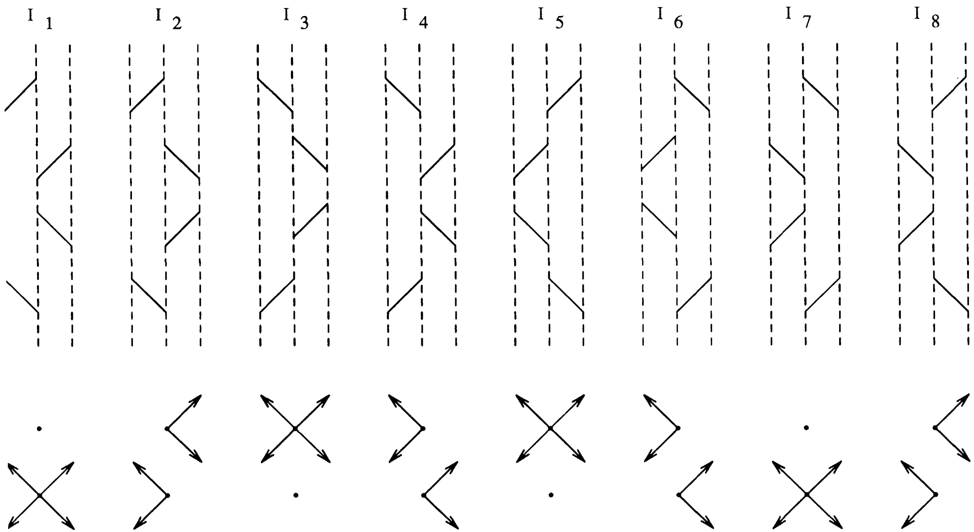


FIG. 13. *Inclusion reference lines.*

the original graph is legal too. Similar comments can be made regarding reference lines of type D_2, D_4, D_5, D_7, I_1, I_3, I_5, and I_7. We will now develop an algorithm to handle reference lines of type D_1, D_3, D_6, D_8, I_2, I_4, I_6, and I_8. In most of these cases, the wiring will be modified and then some vertices of the constraint graph will be selected. Finally the removal of the columns with selected vertices will result in a graph with the property that none of its vertices has two edges adjacent to a single column. The procedure involves a detailed case study which is summarized by the following algorithm.

ALGORITHM MODIFY
Input: Wiring layout produced by Algorithm Wire Nets.

Output: A new wiring with its modified constraint graph and a set of selected vertices.

1. Generate the diagonal diagram, delete all half diagonals and add necessary dummy diagonals. Determine the constraint graph and mark all reference lines of type D_1, D_3, D_6, D_8, I_2, I_4, I_6, I_8 as active.
2. For each inactive reference line, select the vertex of degree 0 if the other vertex has two edges adjacent to a single column. Remove selected columns.
3. Handle type I_2 active reference lines as follows. Let $L_j, L_{j-2}, \cdots, L_{j-2k}$ be a maximal consecutive set of active I_2's. We want to modify every other L_i starting with L_j in a way that depends on the type of its left neighbor L_{i-1}. All the cases that can arise are shown in Fig. 14 with the corresponding modifications. In each such case, a vertex (of degree 0) of L_{i-1} is selected, and a vertex of L_i is selected if the other vertex has two edges adjacent to a column. The columns of selected vertices are removed. Handle type I_6 reference lines in a similar fashion.
4. Handle type I_4 active reference lines as shown in Fig. 15. Select v_{2i} and remove corresponding column. Handle type I_8 similarly.

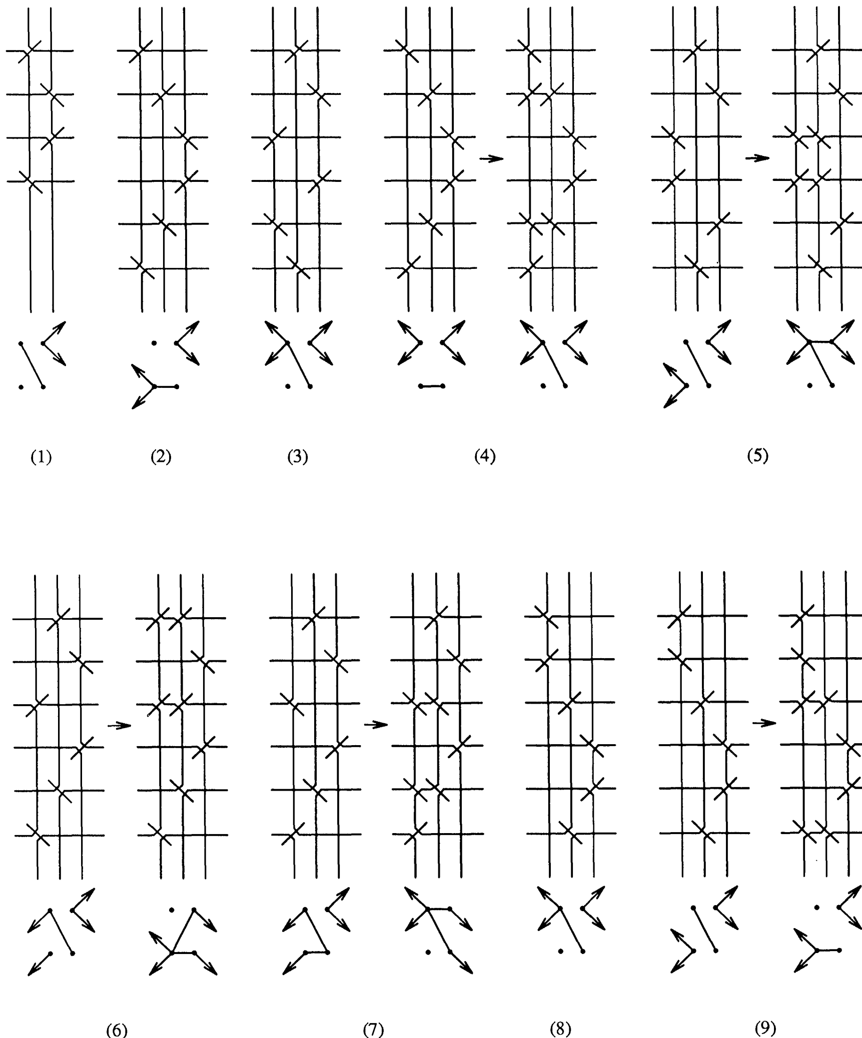


FIG. 14. *Transformations on type I_2 reference lines.*

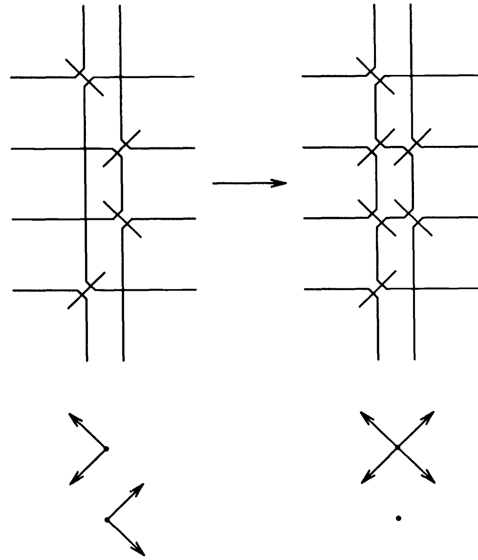


FIG. 15. *Transformations on type I_4 reference lines.*

5. Handle active type D_1 as shown in Fig. 16. Select v_{2i-1} and remove edges between L_i and its neighbors. In Fig. 17 a maximal set of consecutive D_1's is considered. $L_i, L_{i+1}, \cdots, L_k$ are all of type D_1. Modify as shown and select all the odd vertices of $L_i - L_k$. As before selected columns are removed. Repeat the same procedure for types D_3, D_6, and D_8.

We can check that whenever a vertex is selected and the corresponding column removed, the constraint graph is legal if the remaining graph is. Moreover, at the completion of the algorithm, the remaining graph has no vertices with two edges adjacent to a single column. Therefore the overall modified constraint graph is legal.

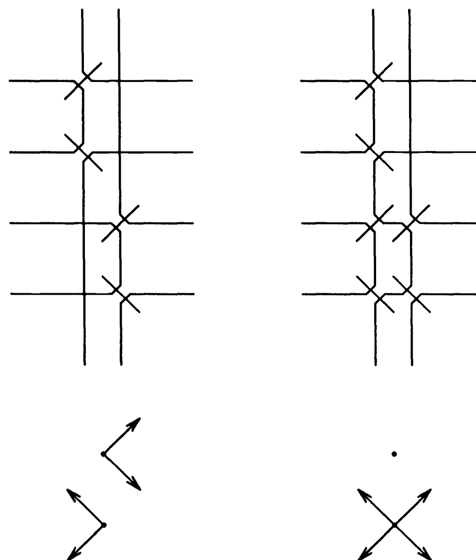


FIG. 16. *Transformations on type D_1 reference lines.*

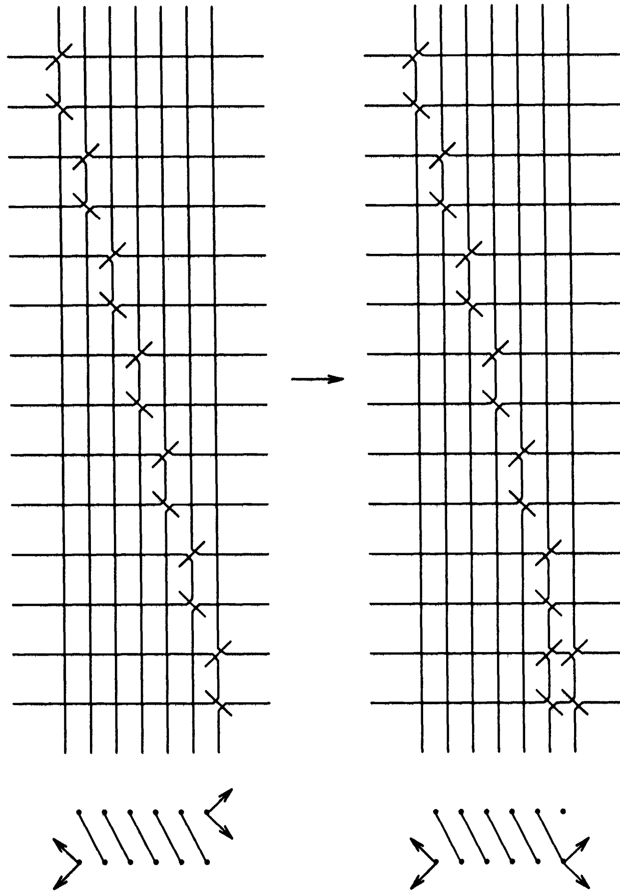


FIG. 17. *Maximal set of consecutive D_1's.*

If we return to the example of Fig. 4, then the routing produced by the algorithm of the previous section is given in Fig. 9. The layer assignment algorithm will change the wiring of N_{16} and N_{21} (Fig. 18) and the final layout is shown in Fig. 19.

THEOREM 2. *Given an instance of the channel routing problem, it is possible to determine a three-layer assignment of the routing layout in time $O(\log n)$ time with $n/\log n$ processors on the* CREW PRAM *model.*

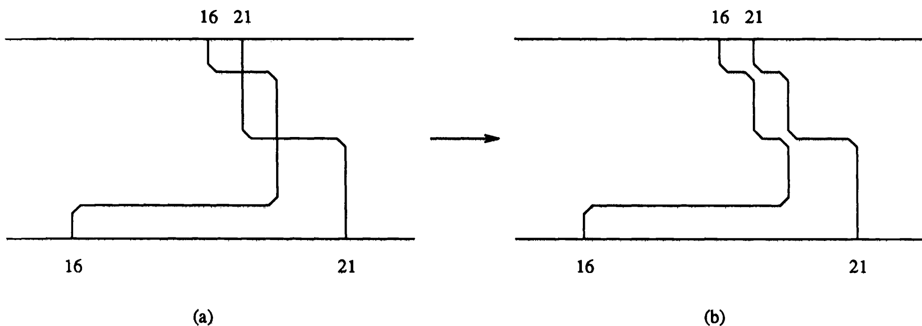


FIG. 18. *Changes in the wiring of N_{16} and N_{21}.*

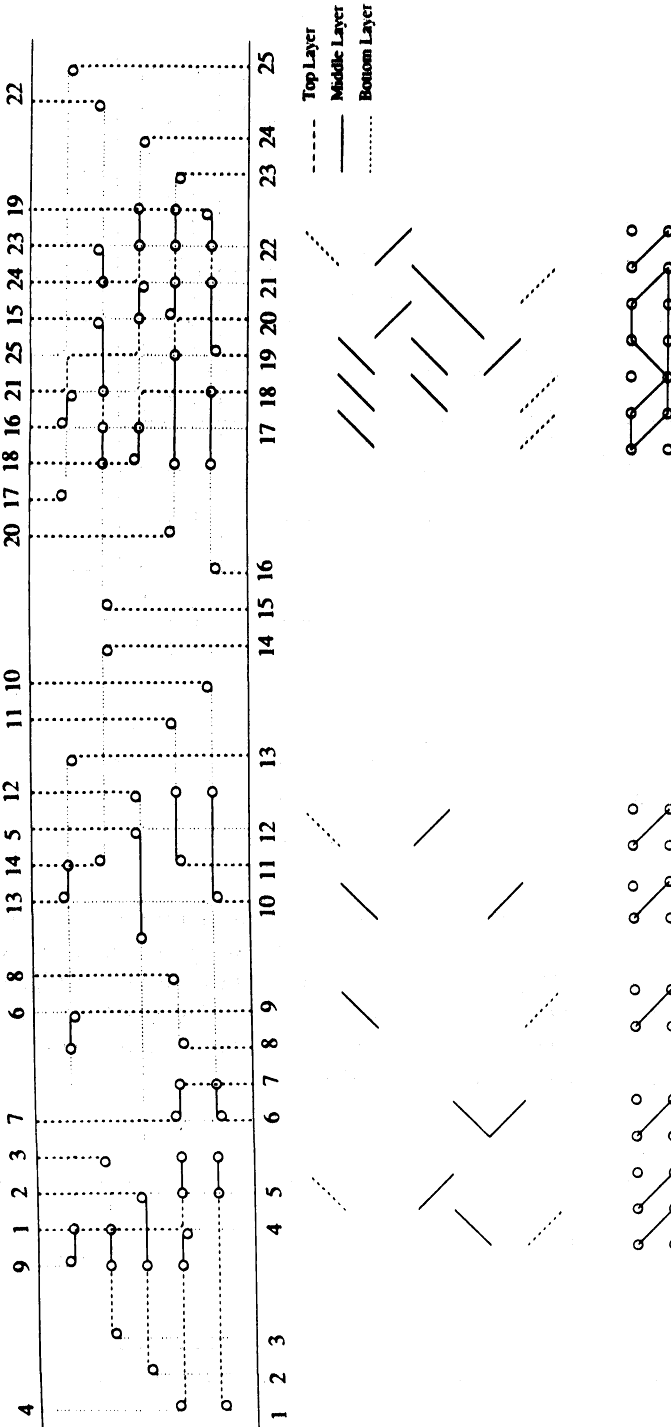


FIG. 19. (a) *The final layout after the modification of layer assignment algorithm,* (b) *its corresponding diagonal diagram, and* (c) *its corresponding constraint graph.*

Acknowledgment. The authors thank one of the referees for bringing reference [DS] to their attention.

REFERENCES

[BSV] O. Berkman, B. Schieber, and U. Vishkin, *Some doubly logarithmic optimal algorithms based on finding all nearest smaller values*, UMIACS-TR-88-79, University of Maryland, College Park, MD, 1988.

[BB] M. Brady and D. Brown, *VLSI routing: Four layers suffice*, in Advances in Computing Research 2 (VLSI Theory), F. Preparata, ed., JAI Press, Greenwich, CT, 1984, pp. 245–257.

[C] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[CV] R. Cole and U. Vishkin, *Approximate and exact parallel scheduling with application to list, tree and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, October 1986, IEEE Computer Society, Washington DC, 1986, pp. 478–491.

[DS] E. Dekel and S. Sahni, *Parallel scheduling algorithms*, Oper. Res., 31 (1983), pp. 24–49.

[Detal] D. Dolev, K. Karplus, A. Seigel, A. Strong, and J. Ullman, *Optimal wiring between rectangles*, in Proc. 13th Annual ACM Symposium on Theory of Computing, May 1981, Association for Computing Machinery, New York, 1981, pp. 312–317.

[KR] R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, Report No. UCB/CSD 88/408, University of California, Berkeley, CA, 1988.

[LP] W. Lipski and F. Preparata, *A unified approach to layout wirability*, Math. Systems Theory, 19 (1987), pp. 189–203.

[L] W. Lipski, *On the structure of three-layer wirable layouts*, in Advances in Computing Research 2 (VLSI Theory), F. Preparata, ed., JAI Press, Greenwich, CT, 1984, pp. 231–243.

[MP] K. Melhorn and F. Preparata, *Routing through a rectangle*, J. Assoc. Comput. Mach., 33 (1986), pp. 60–85.

[O] T. Ohtsuki, *Layout Design and Verification*, Advances in CAD for VLSI, Vol. 4, North-Holland, Amsterdam, 1986.

[P] R. Pinter, *River routing: Methodology and analysis*, in Proc. Third CalTech Conference on VLSI, March 1983, pp. 141–163.

[PL] F. Preparata and W. Lipski, *Optimal three-layer channel routing*, IEEE Trans. Comput., 33 (1984), pp. 427–437.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 246-258, April 1991

© 1991 Society for Industrial and Applied Mathematics
004

SOME OBSERVATIONS ON SEPARATING COMPLEXITY CLASSES*

RONALD V. BOOK†

Abstract. Cai [*J. Comput. System Sci.*, 38 (1989), pp. 68-85] proved that for almost every set A, $\mathrm{PH}(A) \neq \mathrm{PSPACE}(A)$. For every set A, there is a restricted relativization of the class PSPACE (A), denoted PQH (A), with the property that PQH (A) lies between PH (A) and PSPACE (A) (as was previously studied in [*Theoret. Comput. Sci.*, 15 (1981), pp. 41-50] and [*Theoret. Comput. Sci.*, 40 (1985), pp. 237-243]). It is shown here that $\mathrm{PH} \neq \mathrm{PSPACE}$ if and only if, for almost every set A, $\mathrm{PH}(A) \neq \mathrm{PQH}(A)$. Cai's proof shows that for almost every set A, $\mathrm{PQH}(A) \neq \mathrm{PSPACE}(A)$, so that for almost every set A, PQH (A) bounds PSPACE (A) away from PH (A).

Bennett and Gill [*SIAM J. Comput.*, 10 (1981), pp. 96-113] proved that for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP}(A)$. For every set A, there is a restricted relativization of the class NP (A) (previously studied in [*SIAM J. Comput.*, 13 (1984), pp. 461-487]), denoted $\mathrm{NP_B}(A)$, with the property that $\mathrm{NP_B}(A)$ lies between P (A) and NP (A). It is known [*SIAM J. Comput.*, 13 (1984), pp. 461-487] that $\mathrm{P} \neq \mathrm{NP}$ if and only if there exists a set A such that $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$. It is shown here that $\mathrm{BPP} \neq \mathrm{AM}$ if and only if for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$. A result of Kurtz [*Inform. and Control*, 57 (1983), pp. 40-47] is used to show that for almost every set A, $\mathrm{NP_B}(A)$ bounds NP (A) away from P (A).

In addition, it is shown that membership in PSPACE can be characterized in terms of the PQH ( )-operator and membership in AM can be characterized in terms of the $\mathrm{NP_B}$ ( )-operator.

Other results involving quantitative restrictions on access to information from oracles and qualitative restrictions on the use of such information are presented.

Key words. complexity classes, relativizations, polynomial time, polynomial space, P, NP, BPP, AM, PSPACE, the polynomial-time hierarchy, the BP-operator, uniform witnesses

AMS(MOS) subject classifications. 68Q15, 03D15

1. Introduction. Recently, there have been a number of results regarding properties of complexity classes relative to "almost every" oracle set. Of particular interest are the following results:

(a) Bennett and Gill [BG81] showed that for almost every set A, the classes P (A), NP (A), co-NP (A), and PSPACE (A) are pairwise distinct.

(b) Cai [Ca86], [Ca89a] showed that for almost every set A, $\mathrm{PH}(A) \neq \mathrm{PSPACE}(A)$.

As yet, no one has been able to prove that results such as these can be used to establish similar properties of the unrelativized classes. The purpose of this paper is to present some observations that suggest that certain results of this type (i.e., separation results based on the use of random oracles) will *not* provide information about the separation of the corresponding unrelativized classes.

For every set A, there is a restricted relativization PQH (A) of the class PSPACE (A) (previously studied in [BW81], [BBS85]), where $\mathrm{PQH}(A) = \mathrm{PH}(A \oplus QBF)$, with the property that $\mathrm{PH}(A) \subseteq \mathrm{PQH}(A) \subseteq \mathrm{PSPACE}(A)$. It is shown that the following points hold.

THEOREM A.

(a) $\mathrm{PH} \neq \mathrm{PSPACE}$ *if and only if for almost every set* A, $\mathrm{PH}(A) \neq \mathrm{PQH}(A)$.

(b) *For almost every set* A, $\mathrm{PQH}(A) \neq \mathrm{PSPACE}(A)$.

* Received by the editors October 18, 1989; accepted for publication (in revised form) April 26, 1990. Some of these results were announced at the 5th IEEE Conference on Structure in Complexity Theory, Barcelona, Spain, July 1990. This research was supported in part by National Science Foundation grants CCR86-11980 and CCR89-13584.

† Department of Mathematics, University of California, Santa Barbara, California 93106.

It is well known that PH $\neq$ PSPACE if and only if some set S that is ($\leqq_m^{\mathrm{P}}$- or $\leqq_T^{\mathrm{P}}$- or $\leqq_T^{\mathrm{PH}}$-) complete for PSPACE fails to be in PH. Since for every set A, $A \in \mathrm{PH}$ if and only if for almost every set B, $A \in \mathrm{PH}(B)$, such a set S is a "uniform witness" to the separation of PH and PSPACE if for almost every set C, $S \in \mathrm{PQH}(C) - \mathrm{PH}(C)$. This suggests that S is not oracle-dependent. (The reader should note that Cai's witnesses to the separation of PH (A) and PSPACE (A) are oracle-dependent.)

In addition, it is shown that for every set A, there is a restricted relativization of the class NP (A) (previously studied in [BLS84]), denoted $\mathrm{NP_B}(A)$, with the property that $\mathrm{P}(A) \subseteq \mathrm{NP_B}(A) \subseteq \mathrm{NP}(A)$ and such that the following points hold.

THEOREM B. (a) BPP $\neq$ AM *if and only if for almost every set* A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$.

(b) *For almost every set* A, $\mathrm{NP_B}(A) \neq \mathrm{NP}(A)$.

It is well known that BPP $\neq$ AM if and only if some NP-complete set S is not in BPP (and that $P \neq$ NP if and only if some NP-complete set is not in P). Since for every set A, $A \in \mathrm{BPP}$ if and only if for almost every set B, $A \in \mathrm{P}(B)$, such a set S is a "uniform witness" to the separation of BPP and AM if for almost every set C, $S \in \mathrm{NP_B}(C) - \mathrm{P}(C)$.

The reducibilities studied by Bennett and Gill and by Cai are Turing reducibilities whose access to the oracle is restricted only by the running times or work space of the oracle machines used to implement them. In the case of polynomial time-bounded computation, such reducibilities are not known to characterize membership in the classes P, NP, or co-NP; rather, they can be used to characterize membership in the classes BPP, AM, co-AM, and PH. In the case of polynomial space-bounded reducibilities, such reducibilities are known to characterize membership in the class PSPACE.

The results presented here provide evidence for the thesis that theorems about complexity classes relative to random oracles are results about the quantitative restrictions on access to information and the qualitative restrictions on the use of such information. If the access to information from the random oracle and the use made of such information are restricted so as to be essentially the same for both of the relativized classes, then one can make conclusions about the comparison of the unrelativized classes. This is seen from the result that PH $\neq$ PSPACE if and only if for almost every set A, $\mathrm{PH}(A) \neq \mathrm{PQH}(A)$, since it is known that for every set A, $\mathrm{PQH}(A) = \mathrm{PH}(A \oplus QBF)$. The intrinsic characterizations of the unrelativized complexity classes in terms of the appropriate reducibilities and restricted relativizations provide one method of specifying the restrictions on the access and use of information. (Indeed, the study of quantitative restrictions on the access and use of information provided by oracles in relativized computation has been the subject of extensive investigations; see [Bo89] for an overview.)

(The reader should note that in their study of the isomorphism problem for NP-complete sets, Kurtz, Mahaney, and Royer [KMR89] take a somewhat different viewpoint regarding the use of relativizations.)

Three notions are interwoven here: intrinsic characterization theorems for membership in the polynomial complexity classes, where in each case the characterizations involve reducibilities to "almost every oracle set"; known results about relations among these classes that are based on certain (sometimes restricted) relativizations; and conclusions about the relations among these classes that can be obtained by the use of uniform witnesses for proving the separation of the restricted relativizations of these classes. None of the results presented here is technically difficult since all of them depend on results from the literature. The framework described here may be useful for obtaining a better understanding of the implications of certain technical results.

The paper is organized in the following way. Some basic definitions and background are presented in § 2. Both parts of Theorem A, as well as a number of related results, are developed in § 3. The topic of § 4 is the question of whether, for almost every oracle set, the polynomial-time hierarchy relative to that set is infinite. Both parts of Theorem B, as well as a number of related results, are developed in § 5. The topic of § 6 is the question of whether Cai's result [Ca86], [Ca87] that for almost every A, the Boolean hierarchy relative to A is infinite, provides evidence that the unrelativized Boolean hierarchy is infinite. Section 7 contains some remarks.

2. Preliminaries. In this paper the alphabet is fixed to be $\Sigma = \{0, 1\}$. The set of all strings over Σ is denoted by Σ^*. A linear order $<$ for strings is assumed and we write $\Sigma^* = \{w_0, w_1, w_2, w_3, \cdots\}$ where $w_0 < w_1 < w_2 < \cdots$. The collection of all one-way infinite sequences over Σ is denoted Σ^ω.

The length of a string x will be denoted by $|x|$. The cardinality of a set S will be denoted by $\|S\|$. For a set S, the characteristic function of S is denoted by χ_S.

We assume a pairing function $\langle\, ,\rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$ that is computable in polynomial time and has inverses computable in polynomial time.

For sets $A, B \subseteq \Sigma^*$, the *join* of A and B, $A \oplus B$, is defined to be the set $\{0x, 1y \mid x \in A, y \in B\}$.

It is assumed that the reader is familiar with the well-studied complexity classes such as P, NP, BPP, and PSPACE, and the properties of the polynomial-time hierarchy. For more information about properties of these complexity classes, see the textbook by Balcázar, Díaz, and Gabarró [BDG88].

Let $B \subseteq \Sigma^*$. The characteristic sequence $\alpha_B = b_0 b_1 \cdots$ of B is an element of Σ^ω such that $b_n = 1$ if and only if $w_n \in B$. For an element $\alpha \in \Sigma^\omega$, B_α is the set with characteristic sequence α.

Define $\lambda(0) = \lambda(1) = \frac{1}{2}$ on Σ, making Σ into a probability space. By taking the completion of the infinite product on this probability space, one obtains the measure λ on Σ^ω. The measure ν is defined in such a way that for a measurable class $\mathbf{C}$ of sets, $\nu(\mathbf{C})$ is a real number in the interval $[0, 1]$; this measure on Σ^ω corresponds to the measure λ on Σ, which can be interpreted as the Lebesgue measure on the interval $[0, 1]$ if one identifies the element in Σ^ω with the real number in $[0, 1]$ in the usual way. That is, for a class $\mathbf{C}$ of sets, $\nu(\mathbf{C}) = \lambda(\{\alpha \in \Sigma^\omega \mid B_\alpha \in \mathbf{C}\})$. In structural complexity theory, virtually all classes of languages studied in the context of open problems such as P = ? NP, PH = ? PSPACE, etc., are measurable and are closed under finite variation. Thus, we can use the 0-1 Law from probability theory due to Kolmogorov [Ko33] in the same way that Bennett and Gill [BG81] used it. For our purposes, this means that every suitable class of languages has measure that is either 0 or 1. This provides justification for saying that for a class $\mathbf{C}$ of sets such that $\nu(\mathbf{C}) \neq 0$, the property identifying $\mathbf{C}$ holds for *almost every set.* (For additional explanation, see the work of Lutz [Lu90].)

For an oracle machine M, $L(M, A)$ denotes the set of strings accepted by M relative to oracle set A, and $L(M)$ denotes the set of strings accepted by M when no oracle queries are allowed by M. Recall that set A is Turing-reducible to set B in polynomial time, written $A \leqq_T^{\mathrm{P}} B$, if $A \in \mathrm{P}(B)$. Also, we consider Turing reducibility computed nondeterministically in polynomial time: $A \leqq_T^{\mathrm{NP}} B$ if and only if $A \in \mathrm{NP}(B)$.

Let QBF denote the collection of quantified Boolean formulas that are satisfiable; it is well known that QBF is $\leqq_m^{\mathrm{P}}$-complete for PSPACE. In the present paper, no specific properties of QBF, other than its $\leqq_m^{\mathrm{P}}$-completeness for PSPACE, are needed.

Thus, QBF may be thought of as denoting any set that is $\leqq_m^P$-complete (or $\leqq_T^P$- or $\leqq_T^{NP}$- or $\leqq_T^{PH}$-complete) for PSPACE.

Let SAT denote the collection of conjunctive normal form formulas that are satisfiable; it is well known that SAT is $\leqq_m^P$-complete for NP. In the present paper, no specific properties of SAT, other than its $\leqq_m^P$-completeness for NP are needed. Thus, SAT may be thought of as denoting any set that is $\leqq_m^P$-complete (or $\leqq_T^P$-complete) for NP (an exception is made in § 6 where $\leqq_m^P$-completeness is required).

Generalizing on the properties of the class BPP [Gi77] of languages recognized by polynomial-time probabilistic Turing machines with bounded probability of error, Schöning [Sc87] defined the "BP-operator" in the following way. For any class **C**, $BP \cdot \mathbf{C}$ is the class of sets A such that for some C in **C**, some polynomial $p(n)$, and all $x \in \Sigma^*$,

$$\mathrm{Pr}_{p(|x|)}[y : x \in A \text{ iff } \langle x, y\rangle \in C] > \tfrac{3}{4},$$

where for any predicate P and natural number m, $\mathrm{Pr}_m[y\colon \mathrm{P}(y)]$ is the conditional probability $\mathrm{Pr}_m[\mathrm{P}/\Sigma^m] = 2^{-m} \times \|\{y \mid \mathrm{P}(y) \text{ and } |y| = m\}\|$.

As Schöning observed, the class BPP is $BP \cdot P$. Babai and Moran [BM88] observed that the "Arthur–Merlin" class AM (sometimes denoted AM (2)) can be characterized as $BP \cdot NP$, so that it is the nondeterministic counterpart to BPP.

There are some results that will be useful in various places in this paper and are brought together here.

LEMMA 2.1 [TW89]. (a) $PH = BP \cdot PH = \bigcup_{k \geqq 0} BP \cdot \Sigma_k^P$.

(b) $PSPACE = BP \cdot PSPACE$.

(c) *For every suitable class* **C**, $BP \cdot BP \cdot \mathbf{C} = BP \cdot \mathbf{C}$.

LEMMA 2.2. (a) [BG81] *For every set A, $A \in$ BPP if and only if for almost every set B, $A \in$ P (B).*

(b) [NW88] *For every set A, $A \in$ AM if and only if for almost every set B, $A \in$ NP (B).*

(c) [NW88] *For every set A, $A \in$ PH if and only if for almost every set B, $A \in$ PH (B).*

(d) [Ni90] *For every set A, $A \in$ PSPACE if and only if for almost every set B, $A \in$ PSPACE (B).*

As a corollary of the arguments [NW88], there is the following fact.

COROLLARY 2.3. *For every k and every set A, $A \in BP \cdot \Sigma_k^P$ if and only if for almost every set B, $A \in \Sigma_k^P(B)$.*

3. PH versus PSPACE. Cai [Ca89a] proved that for almost every set A, PH $(A) \neq$ PSPACE (A). (Babai [Ba87] gave a short proof of this fact.) In both proofs the witness to the separation of PSPACE (A) from PH (A) depends on the oracle set A. Careful examination of the proofs shows that a stronger statement was established.

PROPOSITION 3.1. *For every fixed set B and for almost every set A,* PH $(A \oplus B) \neq$ PSPACE $(A \oplus B)$.

This observation will be used in this section and the next. It will be applied in the context of certain restricted relativizations of the class PSPACE that are defined by limiting the amount of access to the oracle set that a polynomial space-bounded oracle machine may have. These restricted relativizations were introduced in [Bo81], [BW81], [BBS85].

For every set A, let PQUERY (A) be the collection of sets L such that L's membership in PSPACE (A) can be witnessed by a deterministic oracle machine that uses at most polynomial work space and is restricted in such a way that there is also a polynomial that bounds, as a function of the size of the input, the number of queries

allowed in any computation. For every set A, let NPQUERY (A) be defined in a similar way but with the change that the machines are nondeterministic. For every set A, let $\Sigma_0^{PQ}(A) = \text{PQUERY}(A)$, $\Sigma_1^{PQ}(A) = \text{NPQUERY}(A)$, $\Sigma_{i+1}^{PQ}(A) = \bigcup \{\text{NPQUERY}(B) \mid B \in \Sigma_i^{PQ}(A)\}$ for each $i \geqq 1$, and $\text{PQH}(A) = \bigcup_i \Sigma_i^{PQ}(A)$. It is clear that for every set A, $\text{PH}(A) \subseteq \text{PQH}(A) \subseteq \text{PSPACE}(A)$ and $\text{PH} \subseteq \text{PSPACE} \subseteq \text{PQH}(A)$.

The first result is concerned with the PH = ?PSPACE problem.

THEOREM 3.2. $\text{PH} \neq \text{PSPACE}$ *if and only if for almost every set* A, $\text{PH}(A) \neq \text{PQH}(A)$ *if and only if there exists a set A such that* $\text{PH}(A) \neq \text{PQH}(A)$.

Proof. It is known [BBS85] that for every set A, $\text{PQH}(A) = \text{PH}(A \oplus QBF)$, and it is known [BW81] that PH = PSPACE if and only if for every set A, $\text{PH}(A) = \text{PQH}(A)$. Hence, PH = PSPACE if and only if for every set A, $QBF \in \text{PH}(A)$. Thus, if for almost every set A, $\text{PH}(A) \neq \text{PQH}(A)$, then there exists a set A such that $\text{PH}(A) \neq \text{PQH}(A)$ and $\text{PH} \neq \text{PSPACE}$.

Suppose that $\text{PH} \neq \text{PSPACE}$. Then QBF is not in PH. As noted in Lemma 2.1(a), $\bigcup_{k \geqq 0} \Sigma_k^P = \text{PH} = \text{BP} \cdot \text{PH} = \bigcup_{k \geqq 0} \text{BP} \cdot \Sigma_k^P$, so $QBF \notin \text{PH}$ implies that for every k, $QBF \in \text{BP} \cdot \Sigma_k^P$. From Corollary 2.3, for every k and every set A, $A \in \text{BP} \cdot \Sigma_k^P$ if and only if for almost every set B, $A \in \Sigma_k^P(B)$. Since for every k, $QBF \notin \text{BP} \cdot \Sigma_k^P$, it follows that for every k and for almost every set A, $QBF \notin \Sigma_k^P(A)$. Hence, for almost every set A, $QBF \notin \bigcup_{k \geqq 0} \Sigma_k^P(A) = \text{PH}(A)$. Thus, for almost every set A, $\text{PH}(A) \neq \text{PQH}(A)$. □

Theorem 3.2 shows that if *there exists* one set A such that $\text{PH}(A) \neq \text{PQH}(A)$, then for *almost every* set A, $\text{PH}(A) \neq \text{PQH}(A)$. By using the 0-1 Law one sees that PH = PSPACE if and only if for almost every set A, $\text{PH}(A) = \text{PQH}(A)$. Thus, if for *almost every* set A, $\text{PH}(A) = \text{PQH}(A)$, then for *every* set A, $\text{PH}(A) = \text{PQH}(A)$.

It is known [LS86], [BBS86] that $\text{PH} \neq \text{PSPACE}$ if and only if there exists a sparse set S such that $\text{PH}(S) \neq \text{PQH}(S)$ if and only if for every sparse set S, $\text{PH}(S) \neq \text{PQH}(S)$. (It is known [BBS85] that for every sparse set S, $\text{PSPACE}(S) = \text{PQH}(S)$.) In Theorem 3.2, the condition "for every" is relaxed to "for almost every" and by doing so the restriction to sparse oracle sets disappears.

Since for every set A, $\text{PQH}(A) = \text{PH}(A \oplus QBF)$ and $\text{PSPACE}(A \oplus QBF) = \text{PSPACE}(A)$, Proposition 3.1 yields the following fact.

COROLLARY 3.3. *For almost every set* A, $\text{PQH}(A) \neq \text{PSPACE}(A)$.

Thus, for every set A, $\text{PQH}(A)$ serves as an "upper bound" or "cover" for $\text{PH}(A)$, and for almost every set A, $\text{PQH}(A)$ strictly separates $\text{PH}(A)$ from $\text{PSPACE}(A)$. Also, $\text{PH} \neq \text{PSPACE}$ if and only if for almost every set A, $\text{PQH}(A)$ lies strictly between $\text{PH}(A)$ and $\text{PSPACE}(A)$.

For an arbitrary set A, how much difference is there between $\text{PQH}(A)$ and $\text{PSPACE}(A)$? Combining the fact that for every set A, $\text{PQH}(A) = \text{PH}(A \oplus QBF)$ and $\text{PSPACE}(A \oplus QBF) = \text{PSPACE}(A)$ with another result of Cai (Theorem 4.2 of [Ca89b]), one obtains the following result.

PROPOSITION 3.4. *For almost every set* A, *for every set* B *such that for all* n, $\|\{x \in B \mid |x| \leqq n\}\| = 2^{n^{o(1)}}$, $\text{PSPACE}(A) \not\subseteq \text{PQH}(A \oplus B)$.

Thus, for almost every set A, the PQH operator must have access to a set of very high density in addition to the set A if it is to recognize all of the sets in $\text{PSPACE}(A)$.

Consider the two classes, $\{A \mid \text{PH}(A) \neq \text{PQH}(A) \neq \text{PSPACE}(A)\}$ and $\{A \mid \text{PH}(A) = \text{PQH}(A) \neq \text{PSPACE}(A)\}$. These two classes have opposite measures, either zero or one, depending on whether PH is equal to PSPACE.

The problem of whether PH is equal to PSPACE is a "uniform" problem in the sense that it depends precisely on whether QBF is in PH. The set QBF is a "uniform witness" to the possible separation of PSPACE from PH.

A set S is a *witness to the separation* of two complexity classes **C** and **D** if S

belongs to the symmetric difference of $\mathbf{C}$ and $\mathbf{D}$. Let R denote some type of relativization of complexity classes which lies between that given by many-one reducibilities and that given by Turing reducibilities. A set S is a *uniform witness to the separation, for almost every set A, of $\mathbf{C}_R(A)$ and $\mathbf{D}_R(A)$, if for almost every set A, S is a witness to the separation of $\mathbf{C}_R(A)$ and $\mathbf{D}_R(A)$.*

Notice that if a set S is a uniform witness to the separation, for almost every set A, of $\mathbf{C}_R(A)$ and $\mathbf{D}_R(A)$, then that separation is not oracle dependent.

In most of the results of the present paper, the reducibility R is Turing reducibility. With only a few exceptions, the pairs of complexity classes that are studied here are known to have the relationship that one is included in the other, say $\mathbf{C} \subseteq \mathbf{D}$, and for every set A, $\mathbf{C}_R(A) \subseteq \mathbf{D}_R(A)$.

(The reader may ask why we define uniform witnesses with respect to *almost every* oracle set instead of with respect to *every* oracle set. For a reducibility R lying between many-one and Turing reducibilities, it is the case that for every set A, $A \in \mathbf{C}_R(A)$, so that there is no set S such that for *every* set A, S is a witness to the separation of $\mathbf{C}_R(A)$ and $\mathbf{D}_R(A)$ (since $S \notin \mathbf{C}_R(S) \Delta \mathbf{D}_R(S)$).)

The proof of Theorem 3.2 yields the following fact.

COROLLARY 3.5. PH $\neq$ PSPACE *if and only if QBF is a uniform witness to the separation, for almost every set A, of* PH (A) *and* PQH (A), *if and only if QBF is a uniform witness to the separation, for almost every oracle set A, of* PH (A) *and* PSPACE (A).

The PQH ( )-operator, as a restriction of the PSPACE ( )-operator, has an unexpected property.

THEOREM 3.6. *For every set A, $A \in$* PSPACE *if and only if for almost every set B, $A \in$* PQUERY (B) *if and only if for almost every set B, $A \in$* NPQUERY (B) *if and only if for almost every set B, $A \in$* PQH (B).

Proof. The argument in one direction follows from the fact that for every set B, PSPACE $\subseteq$ PQUERY $(B) \subseteq$ NPQUERY $(B) \subseteq$ PQH (B).

Recall from the definition that for every choice of A and B, if $A \in$ PQUERY (B), then A's membership in PSPACE (B) can be witnessed by a deterministic oracle machine that uses polynomial work space and is restricted so that it can make at most a polynomial, say $p(n)$, number of oracle queries in any computation. Since in space $p(n)$ one can generate in turn all strings of 0's and 1's of length $p(n)$, this leads immediately to the fact that for every set A, if for almost every set B, $A \in$ PQUERY (B), then $A \in$ PSPACE. Similarly, for every set A, if for almost every set B, $A \in$ NPQUERY (B), then $A \in$ PSPACE. From Lemma 2.2(c), it follows that for every set B, if for almost every set B, $A \in$ PQH $(B) =$ PH $(B \oplus QBF)$, then $A \in$ PH $(QBF) =$ PSPACE. $\square$

Schöning [Sc87] has observed that for appropriate classes $\mathbf{C}$, any predicate in BP $\cdot\, \mathbf{C}$ can be expressed in terms of an alternation of two polynomial-bounded quantifiers applied to a predicate in $\mathbf{C}$; hence, BP $\cdot\, \mathbf{C} \subseteq$ PH $(\mathbf{C})$. It follows that if $\mathbf{C}$ has the property that PH $(\mathbf{C}) = \mathbf{C}$, then BP $\cdot\, \mathbf{C} = \mathbf{C}$. This fact provides some intuition for Theorem 3.6 since for every set B, PH (PQH (B)) $=$ PH (PH $(B \oplus QBF)$) $=$ PH $(B \oplus QBF) =$ PQH (B).

The following fact is obtained by combining Corollary 3.3 and Theorem 3.6, and is interesting in its own right.

COROLLARY 3.7. *For every set A, $A \in$* PSPACE *if and only if for almost every set B, $A \in$* PQH (B). *However, for almost every set B,* PQH $(B) \neq$ PSPACE (B).

A different approach has been taken elsewhere. Many of the combinatorial problems that have been shown to be $\leq_m^{\mathrm{P}}$-complete for PSPACE implicitly involve the

encoding of the transitive closure of some binary operation. An interpretation of transitive closure as an operation on a class of languages was developed earlier [Bo79], and it was shown that for every set A, PSPACE (A) is "weakly transitively closed" while PH (A) = PSPACE (A) (PQH (A) = PSPACE (A)) if and only if PH (A) (respectively, PQH (A)) is weakly transitively closed. Proposition 3.1 can be interpreted as saying that for every B and for almost every set A, PH $(A \oplus B)$ is not weakly transitively closed; hence, for almost every set A, neither PH (A) nor PQH (A) is weakly transitively closed.

4. Is the polynomial-time hierarchy infinite? The polynomial-time hierarchy is not known to extend beyond the class P. Yao [Ya85] proved that there exists a set A such that the polynomial-time hierarchy relative to A is an infinite hierarchy of classes: for every k, $\Sigma_k^P(A) \neq \Sigma_{k+1}^P(A)$. There is an interesting open question that is related to the present study. Suppose that for almost every set A, the polynomial-time hierarchy relative to A is an infinite hierarchy of classes, i.e., for almost every set A, for every k, $\Sigma_k^P(A) \neq \Sigma_{k+1}^P(A)$. Does this imply that the (unrelativized) polynomial-time hierarchy is itself an infinite hierarchy?

Suppose that one could prove that: (1) for almost every set B, for every k, $\Sigma_k^P(B) \neq \Sigma_{k+1}^P(B)$. Then it is likely that the proof would also show that (2) for almost every set B, for every k, $\Sigma_k^P(B \oplus QBF) \neq \Sigma_{k+1}^P(B \oplus QBF)$, which would mean that (3) for almost every set B, for every k, $\Sigma_k^{PQ}(B) \neq \Sigma_{k+1}^{PQ}(B)$.
Thus, if for almost every set B, the classes $\Sigma_1^P(B)$, $\Sigma_2^P(B)$, $\cdots$ form a properly infinite hierarchy, then it is likely that for almost every set B, the classes $\Sigma_1^{PQ}(B)$, $\Sigma_2^{PQ}(B)$, $\cdots$ form a properly infinite hierarchy.

If (1) is considered as evidence that the unrelativized polynomial-time hierarchy is infinite, then (3) ought to give equal evidence that the unrelativized polynomial-query hierarchy is infinite. But for every $k > 0$, Σ_k^{PQ} = PSPACE, that is, the unrelativized polynomial-query hierarchy collapses to PSPACE. There is one result to consider.

For every $k > 0$, let B_k be a set that is $\leqq_m^P$-complete for Σ_k^P.

THEOREM 4.1. *The following are equivalent*: (a) *the polynomial-time hierarchy extends to infinitely many levels*;

(b) *for every* $k > 0$, $B_{k+1} \notin \Sigma_k^P$;

(c) *for every* $k > 0$, $B_{k+1} \notin \mathrm{BP} \cdot \Sigma_k^P$;

(d) *for every* $k > 0$ *and for almost every set* A, $B_{k+1} \notin \Sigma_k^P(A)$.

5. P, NP, BPP, AM. Bennett and Gill [BG81] proved that for almost every set A, the classes P (A), NP (A), co-NP (A), and PSPACE (A) are pairwise distinct. A careful examination of the proof shows that a stronger statement was established.

PROPOSITION 5.1. *For every fixed set B and almost every set A, the classes* P $(A \oplus B)$, NP $(A \oplus B)$, *co*-NP $(A \oplus B)$, *and* PSPACE $(A \oplus B)$ *are pairwise distinct.*

This is a generalization of an observation made by Kurtz [Ku83] in his refutation of the random oracle hypothesis of Bennett and Gill.

While studying the problems of whether the class P is equal to the class NP and whether the class NP is closed under complementation, Book, Long, and Selman [BLS84] developed results about relativizations of the classes P and NP that involve restricted access mechanisms for obtaining information from the oracle set. These results would allow one to make conclusions such as P $\neq$ NP or NP $\neq$ co-NP if one could establish properties such as those of Bennett and Gill about these relativizations.

For oracle machine M, a set A, and a string x, let $Q(M, A, x) = \{y \mid$ there is a computation of M on x relative to A that queries the oracle about y's membership

in $A\}$. For every set A, let $\mathrm{NP_B}(A)=\{L(M,A) \mid$ there is a polynomial q such that for all x, $\|Q(M,A,x)\| \leqq q(|x|)\}$.

The subscript "B" in the notation $\mathrm{NP_B}(\,)$ refers to the fact that for M, A, and x, $\|Q(M,A,x)\|$ is *bounded* in size by a polynomial in $|x|$.

The following facts are immediate from the definitions or were established by Book, Long, and Selman [BLS84].

LEMMA 5.2. (a) *For every set* A, $\mathrm{P}(A) \subseteq \mathrm{NP_B}(A) \subseteq \mathrm{NP}(A)$ *and* $\mathrm{NP} \subseteq \mathrm{NP_B}(A)$.

(b) *For every set* A, $\mathrm{NP_B}(A) \subseteq \mathrm{P}(A \oplus SAT)$.

(c) $\mathrm{P}=\mathrm{NP}$ *if and only if for every set* A, $\mathrm{P}(A)=\mathrm{NP_B}(A)$, *if and only if for every set* A, $SAT \in \mathrm{P}(A)$.

(d) $\mathrm{NP}=co\text{-}\mathrm{NP}$ *if and only if for every set* A, $\mathrm{NP_B}(A)=co\text{-}\mathrm{NP_B}(A)$ *if and only if for every set* A, $\overline{SAT} \in \mathrm{NP_B}(A)$.

Consider the problems of whether P is equal to NP and whether NP is closed under complementation. While one might like to have a result similar to Theorem 3.2 for these problems, only Lemma 5.2 is known. However, a more powerful result, similar in scope to Theorem 3.2, does hold for the problem of whether BPP is equal to AM. (At this point the reader may wish to review the material at the end of § 2.)

THEOREM 5.3. $\mathrm{BPP} \neq \mathrm{AM}$ *if and only if for almost every set* A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$.

Proof. The class BPP is closed under $\leqq_m^{\mathrm{P}}$ so that $\mathrm{SAT} \in \mathrm{BPP}$ if and only if $\mathrm{NP} \subseteq \mathrm{BPP}$. Since $\mathrm{BP} \cdot \mathrm{BPP}=\mathrm{BPP}$, $\mathrm{AM}=\mathrm{BP} \cdot \mathrm{NP}$, and $\mathrm{BPP} \subseteq \mathrm{AM}$, this implies that $\mathrm{BPP} \neq \mathrm{AM}$ if and only if $\mathrm{NP} \not\subseteq \mathrm{BPP}$ if and only if $SAT \notin \mathrm{BPP}$. From the characterization of membership in BPP and the 0-1 Law, $SAT \notin \mathrm{BPP}$ if and only if for almost every set A, $SAT \notin P(A)$. Hence, $\mathrm{BPP} \neq \mathrm{AM}$ if and only if for almost every set A, $SAT \notin \mathrm{P}(A)$. By Lemma 5.2(c), for almost every set A, $SAT \notin \mathrm{P}(A)$ if and only if for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$. □

Thus, a necessary and sufficient condition for P and NP to be different is the *existence* of a set A such that $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$, while a necessary and sufficient condition for BPP and AM to be different is that for *almost every* set A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$.

An argument similar to the proof of Theorem 5.3 yields the following result.

THEOREM 5.4. $\mathrm{AM} \neq co\text{-}\mathrm{AM}$ *if and only if for almost every set* A, $\mathrm{NP_B}(A) \neq co\text{-}\mathrm{NP_B}(A)$ *if and only if for almost every set* A, $\mathrm{NP_B}(A) \neq \mathrm{P}(A \oplus SAT)$.

Theorems 5.3 and 5.4 and their proofs yield the following facts.

COROLLARY 5.5. (a) $\mathrm{BPP} \neq \mathrm{AM}$ *if and only if SAT is a uniform witness to the separation, for almost every set* A, *of* $\mathrm{P}(A)$ *and* $\mathrm{NP_B}(A)$, *if and only if SAT is a uniform witness to the separation, for almost every set* A, *of* $\mathrm{P}(A)$ *and* $\mathrm{NP}(A)$.

(b) $\mathrm{AM} \neq co\text{-}\mathrm{AM}$ *if and only if* $\overline{SAT}$ *is a uniform witness to the separation, for almost every set* A, *of* $\mathrm{NP_B}(A)$ *and co-*$\mathrm{NP_B}(A)$, *if and only if* $\overline{SAT}$ *is a uniform witness to the separation, for almost every set* A, *of* $\mathrm{NP}(A)$ *and co-*$\mathrm{NP}(A)$.

Other sets can serve as uniform witnesses in Corollary 5.5(a) as long as such a set S has the property that $S \in \mathrm{AM}$, and for every set A, $\mathrm{P}(A)=\mathrm{NP_B}(A)$, if and only if $S \in \mathrm{P}(A)$. From the results of Book, Long, and Selman, the latter condition forces S to be a set that is $\leqq_T^{\mathrm{P}}$-hard for NP. Similarly, any set that is $\leqq_T^{\mathrm{P}}$-hard for co-NP can serve as a uniform witness in Corollary 5.5(b).

The proof of Theorem 5.3 suggests a characterization of AM which is different from that given by Nisan and Widgerson.

THEOREM 5.6. *For every set* A, $A \in \mathrm{AM}$ *if and only if for almost every set* B, $A \in \mathrm{NP_B}(B)$.

A characterization of membership in AM due to Tang and Watanabe [TW89] will be used to prove Theorem 5.6.

LEMMA 5.7 [TW89]. *For every set A, $A \in \mathrm{AM}$ if and only if for almost every tally set T, $A \in \mathrm{NP}(T)$.*

Proof of Theorem 5.6. Recall that a tally set can be viewed as a set of nonnegative integers in unary notation while a set over the alphabet $\Sigma = \{0, 1\}$ can be viewed as a set of nonnegative integers in binary notation. For a tally set T, let bin (T) denote the corresponding set over Σ, while for any set $A \subseteq \Sigma^*$, let tally (A) denote the corresponding tally set; notice that for every tally set T, tally (bin $(T)) = T$ and for every set $A \subseteq \Sigma^*$, bin (tally $(A)) = A$. Since a string over Σ representing integer n has length $O(\log n)$ and for any fixed $c > 0$ there are only $2^{c \cdot \log n}$ strings in Σ^* of length $c \cdot \log n$, it is clear that for any tally set T, $\mathrm{NP}(T) = \mathrm{NP_B}(T) \subseteq \mathrm{NP_B}(\mathrm{bin}(T))$.

From Lemma 5.7 one sees that for every set A, if $A \in \mathrm{AM}$, then for almost every tally set T, $A \in \mathrm{NP}(T)$, and, hence, for almost every tally set T, $A \in \mathrm{NP_B}(\mathrm{bin}(T))$; thus, if $A \in \mathrm{AM}$, then for almost every set C, $A \in \mathrm{NP_B}(C)$. On the other hand, for every set C, $\mathrm{NP_B}(C) \subseteq \mathrm{NP}(C)$. Hence, for every set A, if for almost every set C, $A \in \mathrm{NP_B}(C)$, then for almost every set C, $A \in \mathrm{NP}(C)$. Thus, if for almost every set C, $A \in \mathrm{NP_B}(C)$, then $A \in \mathrm{AM}$ by Lemma 2.2(b). □

While to some extent Theorem 5.6 is parallel in form to Theorem 3.6, the reader should note that the proofs involve very different ideas.

Consider what Bennett and Gill proved. For each set A, they defined a set RANGE^A and its complement $\mathrm{CORANGE}^A$; showed that for almost every set A, $\mathrm{RANGE}^A \in \mathrm{NP}(A) \subseteq \mathrm{PSPACE}(A)$, $\mathrm{CORANGE}^A \notin \mathrm{NP}(A)$, $\mathrm{RANGE}^A \notin \mathrm{P}(A)$, and $\mathrm{CORANGE}^A \notin \mathrm{P}(A)$; and they concluded that for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP}(A)$, $\mathrm{NP}(A) \neq \text{co-NP}(A)$, and $\mathrm{NP}(A) \neq \mathrm{PSPACE}(A)$. The proofs depend on witnesses whose specification depends on the oracle. Kurtz made certain observations that indicated that the arguments of Bennett and Gill showed more than was claimed. Generalizing on the observations of Kurtz, one sees that for every set B, the following holds: for almost every set A, $\mathrm{RANGE}^A \notin \mathrm{P}(A \oplus B)$ so that $\mathrm{CORANGE}^A \notin \mathrm{P}(A \oplus B)$ and $\mathrm{NP}(A) \not\subseteq \mathrm{P}(A \oplus B)$; in addition, $\mathrm{RANGE}^A \in \mathrm{NP}(A \oplus B)$ but $\mathrm{CORANGE}^A \notin \mathrm{NP}(A \oplus B)$ so that $\mathrm{P}(A \oplus B) \neq \mathrm{NP}(A \oplus B)$ and $\mathrm{NP}(A \oplus B) \neq \text{co-NP}(A \oplus B)$. In particular, for almost every set A, $\mathrm{RANGE}^A \notin \mathrm{P}(A \oplus SAT)$ so that $\mathrm{CORANGE}^A \notin \mathrm{P}(A \oplus SAT)$ and $\mathrm{NP}(A) \not\subseteq \mathrm{P}(A \oplus SAT)$, and $\mathrm{CORANGE}^A \notin \mathrm{NP}(A \oplus SAT)$ so that $\mathrm{NP}(A \oplus SAT) \neq \text{co-NP}(A \oplus SAT)$. It is clear that for every set A, $\mathrm{NP_B}(A) \subseteq \mathrm{NP}(A) \subseteq \mathrm{NP}(A \oplus SAT)$, $\mathrm{P}(A \oplus SAT) \subseteq \Delta_2^{\mathrm{P}}(A)$, and $\mathrm{P}(A \oplus SAT) \subseteq \mathrm{NP}(A \oplus SAT) \subseteq \Sigma_2^{\mathrm{P}}(A)$. These facts yield the next result.

THEOREM 5.8. *For almost every set A,* (a) $\mathrm{NP_B}(A) \neq \mathrm{NP}(A)$,

(b) [Ku83] $\mathrm{NP}(A) \not\subseteq \mathrm{P}(A \oplus SAT)$,

(c) $\mathrm{P}(A \oplus SAT) \neq \mathrm{NP}(A \oplus SAT)$,

(d) $\mathrm{P}(A \oplus SAT) \neq \Delta_2^{\mathrm{P}}(A)$,

(e) $\Delta_2^{\mathrm{P}}(A) \not\subseteq \mathrm{NP}(A \oplus SAT)$,

(f) $\mathrm{NP}(A \oplus SAT) \neq \Sigma_2^{\mathrm{P}}(A)$.

Thus, for every set A, $\mathrm{NP_B}(A)$ serves as an "upper bound" or "cover" for $\mathrm{P}(A)$, and for almost every set A, $\mathrm{NP_B}(A)$ separates $\mathrm{P}(A)$ from $\mathrm{NP}(A)$. Also, $\mathrm{BPP} \neq \mathrm{AM}$ if and only if for almost every set A, $\mathrm{NP_B}(A)$ lies strictly between $\mathrm{P}(A)$ and $\mathrm{NP}(A)$.

Notice that Theorem 5.6 does not address the possibility of $\mathrm{P}(A \oplus SAT)$ being included in $\mathrm{NP}(A)$, the possibility that $\mathrm{NP_B}(A) \neq \mathrm{P}(A \oplus SAT)$, or the possibility that $\mathrm{P}(A) \neq \mathrm{NP_B}(A)$.

The following fact is obtained from Theorem 5.6, Lemma 2.2(b), and Theorem 5.8(a), and is of interest in its own right (see [BLS84]).

COROLLARY 5.9. *For every set A, $A \in \mathrm{AM}$ if and only if for almost every set C,*

$A \in \mathrm{NP}(C)$ if and only if for almost every set C, $A \in \mathrm{NP_B}(C)$. However, for almost every set C, $\mathrm{NP_B}(C) \neq \mathrm{NP}(C)$.

One interpretation of Corollary 5.9 is that for every set A, $\mathrm{NP}(A) - \mathrm{NP_B}(A)$ is very "thin." Since $\mathrm{NP}(A) - \mathrm{NP_B}(A)$ is countable, it has measure 0, so that discussing "thinness" in a formal manner does not appear to be helpful. However, Corollary 5.9 shows that $\{A \mid \mathrm{NP}(A) \neq \mathrm{NP_B}(A)\}$ is a set of measure 1.

Corollary 5.9 states that membership in AM can be characterized by using the operator NP( ) or by using the restricted operator $\mathrm{NP_B}$( ). However, for almost every set B, the operator NP( ) has more power than the restricted operator $\mathrm{NP_B}$( )! No claim is made for the existence of a uniform witness in the separation for almost every oracle set C of $\mathrm{NP_B}(C)$ and $\mathrm{NP}(C)$; the proof is based on the results of Bennett and Gill, whose separation results are oracle dependent.

The operator $\mathrm{NP_B}$( ) has the property that P = NP if and only if for every set A, $\mathrm{P}(A) = \mathrm{NP_B}(A)$. Thus, if there exists a set A_1 such that $\mathrm{P}(A_1) \neq \mathrm{NP_B}(A_1)$, then $P \neq \mathrm{NP}$. It is conceivable that $\mathrm{P} \neq \mathrm{NP}$, but for almost every set A_2, $\mathrm{P}(A_2) = \mathrm{NP_B}(A_2)$ so that BPP = AM; this situation has not been ruled out.

Another property of $\mathrm{NP_B}$( ) is that NP = co-NP if and only if for every set A, $\overline{SAT} \in \mathrm{NP_B}(A)$ if and only if for every set A, $\mathrm{NP_B}(A) = \mathrm{P}(A \oplus SAT)$. It is conceivable that NP $\neq$ co-NP, but for almost every set A_2, $\mathrm{NP_B}(A_2) = \text{co-}\mathrm{NP_B}(A_2)$ so that AM = co-AM; this situation has not been ruled out.

Notice that BPP $\neq$ AM if and only if for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A) \neq \mathrm{NP}(A)$, and AM $\neq$ co-AM if and only if for almost every set A, $\mathrm{P}(A) \neq \mathrm{NP_B}(A) \neq \mathrm{P}(A \oplus SAT)$.

6. The Boolean hierarchy on NP. There has been a great deal of interest in the Boolean hierarchy on NP. There are several different definitions but all are equivalent in the sense that in each case the union of the classes so defined is the Boolean closure of the class NP; equivalently it is the class of sets that are bounded truth-table reducible in polynomial time to *SAT*, where *SAT* denotes any set that is $\leqq_m^{\mathrm{P}}$-complete for NP. It is not known whether there are any classes in the hierarchy other than NP. Kadin [Ka88] showed that if the Boolean hierarchy extends to only finitely many levels, then the polynomial-time hierarchy extends to only finitely many levels. Relativizations of the Boolean hierarchy on NP have been studied and it is known [Ca et al88] that there is a set relative to which the Boolean hierarchy is infinite. In fact, Cai [Ca86], [Ca87] has shown that for almost every set A, the Boolean hierarchy relative to A is infinite. More formally, for every set A and for every $k > 0$, let $B_k(A)$ denote the kth class in the Boolean hierarchy on $\mathrm{NP}(A)$; then $B_k(A) \neq B_{k+1}(A)$.

The following observation is based on Cai's proof.

PROPOSITION 6.1. *For every fixed set C, for almost every set A, and for every $k > 0$, $B_k(A \oplus C) \neq B_{k+1}(A \oplus C)$.*

The next fact follows immediately from Proposition 6.1.

PROPOSITION 6.2. *For almost every set A and every $k > 0$, $B_k(A \oplus QBF) \neq B_{k+1}(A \oplus QBF)$.*

The Boolean hierarchy on *QBF* is the union over all k of the classes $B_k(QBF)$ or, equivalently, the class of sets that are bounded truth-table reducible in polynomial time to *QBF*, i.e., the class PSPACE.

If Proposition 6.1 is evidence that the Boolean hierarchy on NP is infinite, then Proposition 6.2 ought to give equal evidence that for every $k > 0$, $B_k(QBF) \neq B_{k+1}(QBF)$, that is, the Boolean hierarchy based on PSPACE is infinite. But for every

$k > 0$, $B_k(QBF) = \text{PSPACE}$, so that the Boolean hierarchy based on QBF has just one level.

Thus, it is difficult to see how to use the fact that for almost every set A, the Boolean hierarchy relative to A is infinite to show that the Boolean hierarchy on NP is infinite.

7. Other comparisons. Consider the problem of whether the class P is equal to the class PSPACE, or whether the class NP is equal to the class PSPACE. Recall that Bennett and Gill showed that for almost every set A, NP (A) is properly included in PSPACE (A). Using arguments similar to the proofs of Theorems 3.2 and 5.3, it is easy to show that:

(a) BPP $\neq$ PSPACE if and only if for almost every set A, $P(A) \neq$ PQUERY (A) if and only if for almost every set A, $QBF \notin$ P (A); and

(b) AM $\neq$ PSPACE if and only if for almost every set A, NP $(A) \neq$ NPQUERY (A) if and only if for almost every set A, $QBF \notin$ NP (A).

The results above discuss possible relationships among the classes BPP, AM, PH, and PSPACE. In order to consider the corresponding relationships between the classes P, NP, PH, and PSPACE, different reducibilities must be considered, reducibilities that can be used to characterize membership in these classes.

For two sets A, B, define $A \leqq^{\text{P}}_{(\log n-T)} B$ if there exists a deterministic polynomial-time oracle machine M that recognizes A relative to B, that runs in polynomial time, and that is restricted so that for some constant c and all n, any computation of M on an input of length n can have at most $c \cdot \log n$ oracle queries. It is shown in [TB91] that for any set A, $A \in$ P if and only if for almost every set B, $A \leqq^{\text{P}}_{(\log n-T)} B$.

For two sets A, B, define $A \leqq^{\text{NP}}_{(\log n-tt)} B$ if there exist a nondeterministic single-valued transducer G that runs in polynomial time and a nondeterministic acceptor E that runs in polynomial time with the following properties:

(i) there is a constant c such that for all n, on an input of size n, any computation of G that produces output yields a string of at most $c \cdot \log n$ strings;

(ii) for all inputs x, $x \in A$ if and only if E accepts $\langle x, y \rangle$, where y is the string of answers obtained by evaluating the characteristic function of B on each of the strings that G produces on input x. It is shown in [BT90] that for any set A, $A \in$ NP if and only if for almost every set B, $A \leqq^{\text{NP}}_{(\log n-tt)} B$.

For every set A, let $\text{P}_{(\log n)-T}(A) = \{B \mid B \leqq^{\text{P}}_{(\log n-T)} A\}$ and let $\text{NP}_{(\log n)-tt}(A) = \{B \mid B \leqq^{\text{NP}}_{(\log n-tt)} A\}$. It is shown in [BT90] that P $\neq$ NP if and only if for almost every set A, $\text{P}_{(\log n)-T}(A) \neq \text{NP}_{(\log n)-tt}(A)$.

The reader may recall that the polynomial-time hierarchy extends to infinitely many levels if and only if there exists a sparse set S such that the polynomial-time hierarchy relative to S extends to infinitely many levels if and only if for all sparse sets S, the polynomial-time hierarchy relative to S extends to infinitely many levels [BBS86], [LS86]. Tang and Watanabe [TW89] showed that polynomial-time Turing reducibility to almost every tally set characterizes membership in the class BPP and that nondeterministic polynomial-time Turing reducibility to almost every tally set characterizes membership in the class AM. It is easy to see that for every tally set T, PQUERY (T) = NPQUERY (T) = PSPACE (T) = PQH (T); similarly, for every tally set T, $\text{NP}_{\text{B}}(T) = \text{NP}(T)$. Combining these observations with the results above yields the following fact.

THEOREM 7.1. (a) BPP $\neq$ AM *if and only if for almost every tally set* T, P $(T) \neq$ NP (T).

(b) AM $\neq$ *co*-AM *if and only if for almost every tally set* T, NP $(T) \neq$ *co*-NP (T).

(c) BPP $\neq$ PSPACE *if and only if for almost every tally set* T, P $(T) \neq$ PSPACE (T).

(d) AM $\neq$ PSPACE *if and only if for almost every tally set* T, NP $(T) \neq$ PSPACE (T).

Thus, if one could obtain results such as those of Bennett and Gill and of Cai for tally oracle sets, then the unrelativized classes would be separated.

Can one use results about "almost every oracle set" to separate unrelativized complexity classes? The results presented here provide evidence that this can be done if the access to information from the random oracle and the use made of such information are restricted so as to be precisely the same for both of the relativized classes. This is a strong restriction but it may be necessary.

If there is a uniform witness to the separation, for almost every set A, of PQH (A) from PH (A), then PH is not equal to PSPACE; and if there is a uniform witness to the separation, for almost every set A, of $\mathrm{NP_B}(A)$ from P (A), then BPP is not equal to AM (so P is not equal to NP). Results such as this suggest the concept of a "uniformity principle." This idea is consistent with the fact that the problems "is PH equal to PSPACE?" and "is BPP equal to AM?" are uniform in nature since in each case the answer depends on a single uniform witness.

Results such as those of Cai and of Bennett and Gill seem to depend on some notion of a "nonuniformity principle." It is possible that the formulation of such a principle would lead to a better understanding of the essential nature of comparing complexity classes by considering their relativizations. This is an interesting open problem involving the foundations of computer science.

Acknowledgments. Richard Beigel made some constructive comments on earlier versions of this paper. Ker-I Ko's questions motivated § 4. Some of the results in § 3 stem from comments and questions by Alan Selman. It is a pleasure to thank each of these individuals for their assistance.

REFERENCES

[Ba87] L. BABAI, *A random oracle separates* PSPACE *from polynomial hierarchy*, Inform. Process Lett., 26 (1987), pp. 51–53.

[BM88] L. BABAI AND S. MORAN, *Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[BDG88] J. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* I, Springer-Verlag, Berlin, New York, 1988.

[BBS85] J. BALCÁZAR, R. BOOK, AND U. SCHÖNING, *On bounded query machines*, Theoret. Comput. Sci., 40 (1985), pp. 237–243.

[BBS86] ———, *The polynomial-time hierarchy and sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 603–617.

[BG81] D. BENNETT AND J. GILL, *Relative to a random oracle A*, $\mathrm{P}^A \neq \mathrm{NP}^A \neq$ *co*-NP^A *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[Bo79] R. BOOK, *On languages accepted by space-bounded oracle machines*, Acta Inform., 12 (1979), pp. 177–185.

[Bo81] ———, *Bounded query machines: On* NP *and* PSPACE, Theoret. Comput. Sci., 15 (1981), pp. 27–39.

[Bo89] ———, *Restricted relativizations of complexity classes*, in Computational Complexity Theory, J. Hartmanis, ed., Proc. Sympos. Appl. Math., Vol. 38, 1989, pp. 47–74.

[BLS84] R. BOOK, T. LONG, AND A. SELMAN, *Quantitative relativizations of complexity classes*, SIAM J. Comput., 13 (1984), pp. 461–487.

[BT90] R. BOOK AND S. TANG, *Characterizing polynomial complexity classes by reducibilities*, Math. Systems Theory, 23 (1990), pp. 165–174.

[BW81] R. BOOK AND C. WRATHALL, *Bounded query machines: On* NP( ) *and* NPQUERY( ), Theoret. Comput. Sci., 15 (1981), pp. 41–50.

[Ca86] J.-Y. CAI, *On some most probable separations of complexity classes*, Ph.D. thesis, Cornell University, Ithaca, NY, 1986.

[Ca87] J.-Y. CAI, *Probability one separation of the boolean hierarchy*, STACS 87, Lecture Notes in Comput. Sci. 247, Springer-Verlag, Berlin, New York, 1977, pp. 148-158.

[Ca89a] ———, *With probability one, a random oracle separates* PSPACE *from the polynomial-time hierarchy*, J. Comput. System Sci., 38 (1989), pp. 68-85.

[Ca89b] ———, *Lower bounds for constant depth circuits in the presence of help bits*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 532-539.

[Ca et al88] J.-Y. CAI, T. GUNDERMAN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The Boolean hierarchy* I, SIAM J. Comput., 17 (1988), pp. 1232-1252.

[Gi77] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675-695.

[Ka88] J. KADIN, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263-1282.

[Ku83] S. KURTZ, *On the random oracle hypothesis*, Inform. and Control, 57 (1983), pp. 40-47.

[KMR89] S. KURTZ, S. MAHANEY, AND J. ROYER, *The isomorphism conjecture fails relative to a random oracle*, submitted; Extended abstract appears in Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 157-166.

[Ko33] A. N. KOLMOGOROV, *Foundations of Probability Theory*, English translation, Chelsea, New York, 1950.

[LS86] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618-627.

[Lu90] J. LUTZ, *Category and measure in complexity classes*, SIAM J. Comput., 6 (1990), pp. 1100-1131.

[Ni90] N. NISAN, *Pseudorandom generators for space-bounded computation*, in Proc. 22nd ACM Symposium on Theory of Computing, 1990, pp. 204-212.

[NW88] N. NISAN AND A. WIGDERSON, *Hardness vs. randomness*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 2-11.

[Sc87] U. SCHÖNING, *Probabilistic complexity classes and lowness*, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 2-8; also appeared in J. Comput. System Sci., 39 (1989), pp. 84-100.

[TB91] S. TANG AND R. BOOK, *Polynomial reducibilities and almost every oracle set*, Theoret. Comput. Sci., 1991, to appear.

[TW89] S. TANG AND O. WATANABE, *On tally relativizations of* BP-*complexity classes*, SIAM J. Comput., 18 (1989), pp. 449-462.

[Ya85] A. C.-C. YAO, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th IEEE Symposium Foundations of Computer Science, 1985, pp. 1-10.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 259-269, April 1991
© 1991 Society for Industrial and Applied Mathematics
005

AN $O(n \log^2 h)$ TIME ALGORITHM FOR THE THREE-DIMENSIONAL CONVEX HULL PROBLEM*

HERBERT EDELSBRUNNER† AND WEIPING SHI‡

Abstract. An algorithm is presented that constructs the convex hull of a set of n points in three dimensions in worst-case time $O(n \log^2 h)$ and storage $O(n)$, where h is the number of extreme points. This is an improvement of the $O(nh)$ time gift-wrapping algorithm and, if $h = o(2^{\sqrt{\log_2 n}})$, of the $O(n \log n)$ time divide-and-conquer algorithm.

Key words. computational geometry, convex hull, three dimensions, output sensitive

AMS(MOS) subject classifications. 68U05, 52-04, 52A15

1. Introduction. The *convex hull* of a set of points S in three-dimensional space is the smallest convex set that contains S. If S is finite then the convex hull of S is a convex polytope with vertices, edges, and facets making up its boundary. The *convex hull problem* is to determine the points in S that are vertices of this convex polytope (the *extreme points* of S, or ext (S)), possibly together with some ordering and adjacency information. Finding efficient convex hull algorithms is one of the most intensively studied problems in computational geometry (see Preparata and Shamos (1985) and Edelsbrunner (1987) for more information). Table 1.1 summarizes the current best results on the worst-case complexity of the convex hull problem not including the result of this paper. The $\Omega(n \log n)$[1] lower bounds in two and three dimensions are due to Yao (1981) and simple proofs follow from Ben-Or (1983). Matching upper bound in two and three dimensions can be found in Graham (1972) and in Preparata and Hong (1977). The $\Omega(n^{\lfloor d/2 \rfloor})$ lower bound in $d \geqq 4$ dimensions follows from the

TABLE 1.1

The current best results for finding the convex hull, where n is the number of input points, h is the number of extreme points, F is the number of faces of any dimension, and d is assumed to be a fixed constant. The first part of the table shows the best results that do not depend on h and assume the worst case over all values of h. The second part gives the output-sensitive results.

Dimension	Lower bound	Upper bound
2	$\Omega(n \log n)$	$O(n \log n)$
3	$\Omega(n \log n)$	$O(n \log n)$
$d \geqq 4$	$\Omega(n^{\lfloor d/2 \rfloor})$	$O(n^{\lceil d/2 \rceil})$
		$O(n^{\lfloor d/2 \rfloor} \log n)$
2	$\Omega(n \log h)$	$O(n \log h)$
3	$\Omega(n \log h)$	$O(nh)$
$d \geqq 4$	$\Omega(n \log h + F)$	$O(n^2 + F \log h)$

* Received by the editors February 7, 1990; accepted for publication (in revised form) July 20, 1990.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. The research of this author was supported by National Science Foundation grant CCR-8714565.

‡ Coordinated Science Laboratory and Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. The research of this author was supported by the Semiconductor Research Corporation under contract 88-DP-109.

[1] Throughout the paper, log is base 2 unless otherwise specified.

same lower bound on the maximum number of faces of the convex hull of n points (see, e.g., Edelsbrunner (1987)). The $O(n^{\lceil d/2 \rceil})$ upper bound is a careful implementation of the beneath-beyond method due to Seidel (1981) and is also described in Edelsbrunner (1987). The $O(n^{\lfloor d/2 \rfloor} \log n)$ upper bound is due to Seidel (1986). The $\Omega(n \log h)$ lower bounds in two and three dimensions are given by Kirkpatrick and Seidel (1986). The currently best output-sensitive algorithm in three dimensions, which takes $O(nh)$ time, is the gift-wrapping algorithm of Chand and Kapur (1970). In $d \geqq 4$ dimensions, $O(F)$ is the size of the convex hull, and therefore a trivial lower bound. The $O(n^2 + F \log h)$ time algorithm is due to Seidel (1986).

It is interesting to note that the lower bounds of Table 1.1 in two and three dimensions also hold for the weaker problem of finding all extreme points, without any order and adjacency information. In higher dimensions the complexity of this problem is considerably less than that of constructing the convex hull itself. By solving n linear programs the extreme points can be found in $O(n^2)$ time using results of Megiddo (1984), if the number of dimensions is a fixed constant. At this point, it is worth mentioning that in three dimensions the hard part of the convex hull problem is to identify extreme points; thereafter, $O(h \log h)$ time suffices to construct the adjacency and order information. This allows us to be loose in explaining what exactly our algorithm outputs. We will design it such that it produces all extreme points plus the pairs that define edges of the convex hull, but we will ignore the order of edges around vertices. Because the edges define a three-connected planar graph, $O(h)$ time suffices to find the unique embedding (see Hopcroft and Tarjan (1974) and also Kirkpatrick (1987)).

For the two-dimensional convex hull problem, Kirkpatrick and Seidel (1986) give an $O(n \log h)$ time algorithm and prove it is asymptotically optimal if the complexity of the problem is measured in terms of input and output sizes. They also raise the question of whether there exists an $O(n \log h)$ time algorithm for constructing the convex hull in three dimensions. Up to now, the best deterministic three-dimensional convex hull algorithm whose complexity depends on n and h is the $O(nh)$ time "gift-wrapping" method of Chand and Kapur (1970). Using randomization, a concept we do not consider in this paper, Clarkson and Shor (1989) give an algorithm running in expected time $O(n \log h)$.

This paper presents an $O(n \log^2 h)$ worst-case time algorithm for the three-dimensional convex hull problem. Following Kirkpatrick and Seidel (1986), the algorithm uses the approach "marriage before conquest" in which it first determines how the solutions of the subproblems will combine and then proceeds to solve the subproblems recursively. The main idea of the algorithm is to first project S onto two carefully chosen planes. Then we use the $O(n \log h)$ time two-dimensional convex hull algorithm to find the convex hulls for the projected points. The two two-dimensional convex hulls are projections of edge sequences of the convex hull of S. They are used to partition S into subsets in a balanced way. By recursively finding the convex hulls for each of the subsets, we can get the convex hull of S.

Section 2 presents the algorithm, § 3 assesses its complexity, § 4 remarks on problems caused by points not in general position, and § 5 concludes this paper with a brief discussion of the results.

2. The algorithm. Let S be a finite set of points in three-dimensional space. We assume general position, that is, no four points are coplanar and no three points lie on a common vertical plane. This assumption is algorithmically justified by the conceptual perturbation technique of Edelsbrunner and Mücke (1990). With this assumption,

the convex hull of S is a simplicial polytope, that is, every facet is a triangle. In addition, no edge or facet is vertical.

We call the part of the convex hull of S that can be seen from $(0, 0, \infty)$ the *upper hull* of S (see Fig. 2.1). Similarly, the *lower hull* of S is the part of the convex hull that can be seen from $(0, 0, -\infty)$. Thus, both the upper hull and the lower hull are simply connected subsets of the boundary of the convex hull of S. Indeed, the boundary of the convex hull is the union of the upper hull and the lower hull, while the intersection of the upper hull and lower hull is the cycle of edges that admit vertical supporting planes.

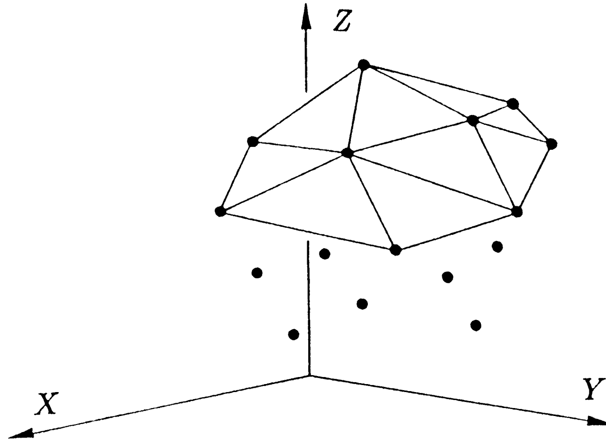


FIG. 2.1. *The upper hull of S.*

The following algorithm shows how to construct the upper hull; the method to construct the lower hull is symmetric. The union can be constructed in $O(h)$ time once we have both the upper and the lower hull. When we describe the procedure we assume that the reader is familiar with the $O(n)$ time three-variable linear programming algorithms of Dyer (1984) or Megiddo (1983), the $O(n)$ time planar ham-sandwich cut algorithm of Megiddo (1985), the $O(n \log h)$ time two-dimensional convex hull algorithm of Kirkpatrick and Seidel (1986), and one of the optimal planar point location algorithms of Kirkpatrick (1983), Cole (1986), Sarnak and Tarjan (1986), and Edelsbrunner, Guibas, and Stolfi (1986).

When we describe the algorithm we use the notational convention that a set primed means the projection of the set onto the XY-plane. For example, $S' = \{(x, y) \mid (x, y, z) \in S\}$. The nondegeneracy assumption that no three points of S lie in a vertical plane implies that no two points lie on a common vertical line which guarantees a bijection between S and S'.

ALGORITHM 3D_UPPER_HULL $(S, \mathcal{B})$.
Input. S is a set of points in space. $\mathcal{B}$ is a simple closed polygonal curve in space whose vertices form a subset of S. $\mathcal{B}'$, the projection of $\mathcal{B}$ onto the XY-plane, is the boundary of a simple polygon and all points of S' lie on the boundary or inside this polygon.
Output. The part of the upper hull of S whose relative boundary is $\mathcal{B}$, represented as an edge-point adjacency list.

Method

if $|S|=3$
0: **then** return $\mathscr{B}$
1: **else** do a 4-division of S' using two intersecting lines in the XY-plane;
2: use three-variable linear programming to find the triangle of the upper hull that intersects the vertical line through the center of the 4-division;
3: project S onto the vertical plane of the first line and compute the two-dimensional upper hull;
4: do the same for the second line;
5: project the two three-dimensional polygonal paths obtained in Steps 3 and 4 onto the XY-plane;
6: recurse inside each connected region of the thus constructed subdivision;
7: return the combination of all recursively computed upper hulls

endif.

End of Algorithm.

Below, we explain in detail each of the steps above.

Step 0. If $|S|=3$, then return $\mathscr{B}$ which, in this case, is a triangle in space.

Step 1. Use the algorithm of Megiddo (1985) to find two lines l_1 and l_2 that divide the XY-plane into four quadrants containing about a quarter of the points each. Because the points in S' are in general position (no three are collinear) we can assume that neither l_1 nor l_2 contains a point of S and that each quadrant contains at least $\lfloor |S'|/4 \rfloor$ points. If the equations of these lines are

$$l_1: a_1x+b_1y+c_1=0 \quad \text{and} \quad l_2: a_2x+b_2y+c_2=0,$$

then the intersection point $p=(p_x, p_y)$ of l_1 and l_2 is given by

$$p_x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \quad \text{and} \quad p_y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}.$$

Let Γ_1 and Γ_2 be the vertical planes that intersect the XY-plane at l_1 and l_2 (see Fig. 2.2).

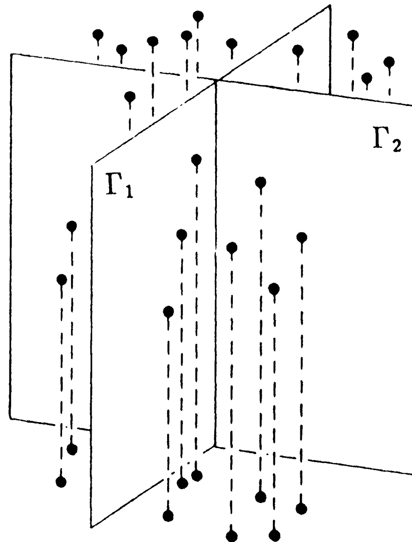


FIG. 2.2. *Cut by two vertical planes,* Γ_1 *and* Γ_2.

Step 2. Use the algorithm of Dyer (1984) or Megiddo (1983) to solve the following three-variable linear programming problem:

$$\text{minimize} \quad \gamma_1 p_x + \gamma_2 p_y + \gamma_3$$

$$\text{subject to} \quad \gamma_1 x_i + \gamma_2 y_i + \gamma_3 \geqq z_i \quad \text{for all } (x_i, y_i, z_i) \in S.$$

The solution to this linear program is a plane $\Gamma: z = \gamma_1 x + \gamma_2 y + \gamma_3$, such that every point of S lies on or below Γ and it has the lowest intersection point with the vertical line through point $p = (p_x, p_y)$ among all those planes. The 4-partition computed in step 1 is such that each quadrant contains at least one point (because $\lfloor |S'|/4 \rfloor \geqq 1$) which implies that, indeed, the vertical line through p intersects the upper hull. We can also assume that p is not collinear with any two points of S' and thus Γ is unique. By our nondegeneracy assumption Γ passes through exactly three points, $a = (a_x, a_y, a_z)$, $b = (b_x, b_y, b_z)$, and $c = (c_x, c_y, c_z)$ of S, and is given by the equation

$$\Gamma: \begin{vmatrix} x & y & z & 1 \\ a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \end{vmatrix} = 0$$

(see Fig. 2.3).

Step 3. Project S onto the plane Γ_1 where the direction of the projection is parallel to $\Gamma \cap \Gamma_2$. Call the obtained point set $S(\Gamma_1)$ and assume a bijection between S and $S(\Gamma_1)$. Using Kirkpatrick and Seidel's algorithm compute the two-dimensional upper hull of $S(\Gamma_1)$ and let H_1 be the corresponding polygonal path in space, that is, the vertices of H_1 are points of S and the projection of H_1 onto Γ_1, along $\Gamma \cap \Gamma_2$, is the upper hull of $S(\Gamma_1)$.

Step 4. Project S onto the plane Γ_2 along the direction parallel to $\Gamma \cap \Gamma_1$. Let the resulting set be $S(\Gamma_2)$ and compute H_2 in complete analogy to H_1 in Step 3.

Step 5. Initialize $H \leftarrow \mathcal{B}$. Add H_1, H_2 and the triangle abc to H; H' is a subdivision of the XY-plane. Discard duplicate edges and edges whose projections lie outside $\mathcal{B}'$.

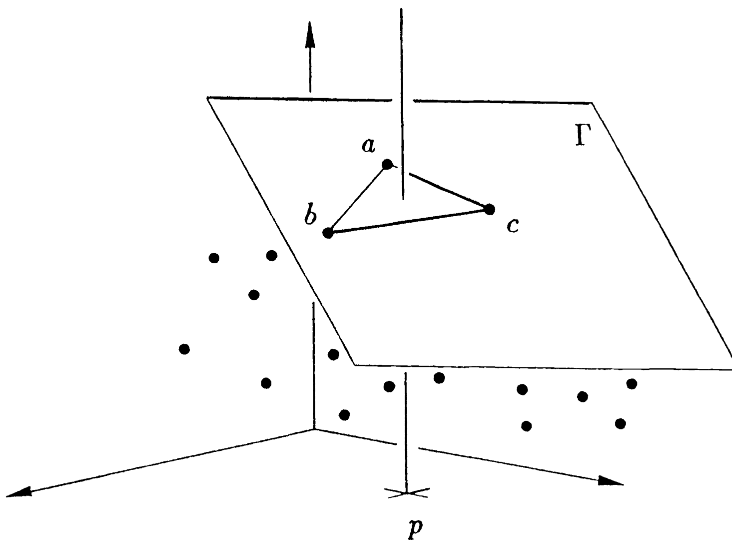


FIG. 2.3. *Use linear programming to find a supporting triangle.*

Build a planar point location data structure based on H'. The data structure can be any one of the optimal methods described by Kirkpatrick (1983), Cole (1986), Sarnak and Tarjan (1986), and Edelsbrunner, Guibas, and Stolfi (1986). Associate each region R_i' of H' with an initially empty set S_i. For every point $q \in S$, decide which region R_i' contains q'; put q into set S_i. We also add points of S to S_i whose projections lie on the boundary of R_i'. Discard points inside $a'b'c'$.

Step 6. For every region R_i' of H' do $H \leftarrow H \cup$ 3D_UPPER_HULL $(S_i, \mathcal{B}_i)$, where $\mathcal{B}_i'$ is the boundary of R_i'.

Step 7. Return H.

Before the first call of the 3D_UPPER_HULL algorithm, we find the convex hull of S' and let $\mathcal{B}$ be the three-dimensional polygonal curve that projects onto its boundary.

Remark. The fairly complicated general point location method in Step 5 can be avoided if we exploit special properties of H_1' and H_2'. The partitioning of S into subsets S_i can be done in linear time since the polygonal paths, H_1' and H_2', are monotone and the two-dimensional convex hull algorithm of Kirkpatrick and Seidel (1986) not only produces H_1 and H_2, but also "sorts" S into $|H_1|$ and $|H_2|$ buckets. We spend a paragraph explaining how this works.

Assume for simplicity that l_1 is the Y-axis and l_2 is the X-axis. Then H_1' is monotone in the Y-direction and H_2' is monotone in the X-direction. Consider the regions of the subdivision formed by H_1' and H_2' that lie to the left of H_1' and below H_2' (see Fig. 2.4 where H_1' consists of the branches labeled N and S and H_2' consists of the branches labeled E and W). Observe that there is a polygonal path Q that is X- and Y-monotone and separates H_1' from H_2' (dotted line in Fig. 2.4); it can be constructed in linear time from either H_1' or H_2'. If a point q lies in region R_i then the vertical line through q meets Q somewhere inside R_i or the horizontal line through q does so. Thus, after doing binary search twice, once in H_1' and once in H_2', we can narrow down the search to at most two regions. If q lies below or to the right of Q then the horizontal line has priority over the vertical line, and vice versa if q lies above or to the left of Q. The two binary searches are done implicitly in the two-dimensional convex hull construction and thus add no extra time to the algorithm.

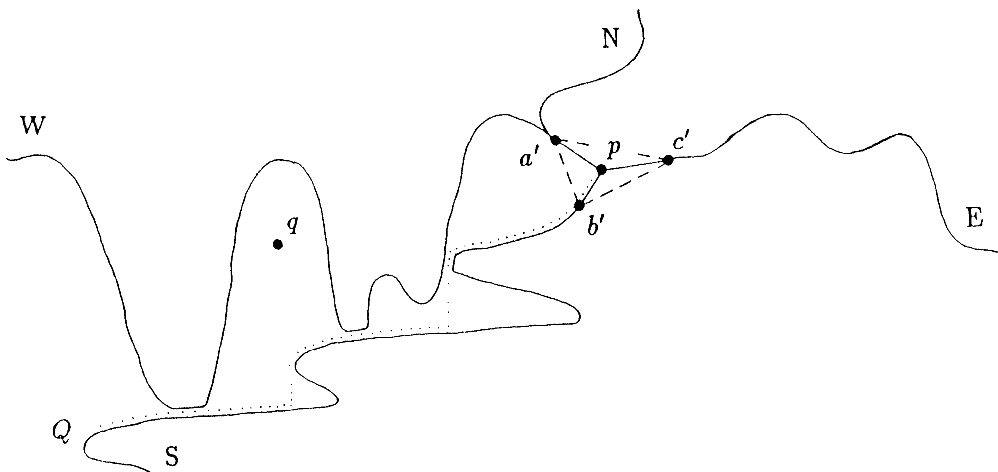


FIG. 2.4. *Projection of* H_1, H_2 *and triangle abc.* H_1' *is decomposed into* N (*North*) *and* S (*South*) *and* H_2' *is decomposed into* E (*East*) *and* W (*West*), *and all four branches are connected to p. Note that some branches share common parts.*

In the remainder of this section, we argue that the algorithm correctly finds the part of the upper hull of S whose projection onto the XY-plane lies in $\mathcal{B}'$. We use induction on the number of points in S. If $|S|=3$, the algorithm is trivially correct. When $|S|>3$, every edge e (and therefore every vertex) on H_1 and H_2 must also be on the upper hull of S since there exists a plane that passes through e and has all other points of S below it. For the same reason triangle abc must be on the upper hull. Points inside triangle $a'b'c'$ cannot be on the upper hull because all these points are under the facet abc and can therefore not be seen from $(0, 0, +\infty)$. It remains to show that by partitioning S into subsets the algorithm does not lose any edges of the upper hull.

Recall that every facet is a triangle because we assume that no four points are coplanar. If there were an edge p_ip_j on the upper hull and $p_i \in S_i$ and $p_j \in S_j$, $i \neq j$, then the projection onto the XY-plane of the edge p_ip_j must cross some edge of H_1', H_2', $\mathcal{B}'$ or triangle $a'b'c'$. We then have at least four points on the same plane, violating the nondegeneracy assumption.

Remark. It is not absolutely necessary that the points are in general position for our algorithm to work. However, with the nondegeneracy assumption it is significantly simpler to describe, to understand, and to code.

3. Analysis. Finally, we are ready to assess the time-complexity of the algorithm. We start by proving a general lemma on the total cost of certain trees. Then we show by a geometric argument that the cost of the algorithm can be modeled by such a tree and finally conclude with the main result of this paper.

DEFINITION. Let $T=(V, E)$ be a rooted tree with a cost function $c: V \to [0, +\infty)$. If there is a constant $\alpha \in (0, 1)$, such that $c(\mu) \leqq \alpha c(\nu)$ for every node μ and its parent ν, then we say the cost function c is *fading* and α is the *fading factor.*

LEMMA 3.1. *In a rooted tree $T=(V, E)$ with fading cost function c and fading factor α, if for each level the sum of the costs of all nodes at this level is bounded by C, then the sum of the costs of all nodes in the tree is at most* $C(\log_{1/\alpha}|V|+O(1))$.

Proof. The proof consists of two steps. We first change the shape of the tree by repeated application of a path compression operation without increasing its total cost nor violating the level cost bound. Then we claim that after changing the shape, the tree has height $\log_{1/\alpha}|V|+O(1)$ and finish the proof.

Path compression operation. Arbitrarily pick a node v from the bottommost level and make it a new child of one of its ancestors.

It is clear that after the application of this operation T is still a tree, c is still fading, and the total cost of the tree did not change. To change the tree we number its levels 0, 1, 2, etc. with the root at level zero. If some level i that is not the bottommost level has fewer than $\lfloor \alpha^{-i} \rfloor$ nodes, then we apply the path compression operation and make v a child of its ancestor at level $i-1$. This is done until level i has at least $\lfloor \alpha^{-i} \rfloor$ nodes, for each but the bottommost level i. The cost of level i increases only if in the end it has exactly $\lfloor \alpha^{-i} \rfloor$ nodes (except for the bottommost level which may have fewer nodes). These $\lfloor \alpha^{-i} \rfloor$ nodes have a cost that does not exceed the cost of the root and is therefore not greater than C.

Now consider the total number of nodes in the resulting tree, where l is the height of the tree (so level l is the bottommost level). We have

$$|V| \geqq \sum_{i=0}^{l-1} \lfloor \alpha^{-i} \rfloor \geqq \sum_{i=0}^{l-1} \alpha^{-i} - l = \frac{(1/\alpha)^l - 1}{1/\alpha - 1} - l \geqq \frac{(1/\alpha)^l - 1}{1/\alpha - 1} - |V|.$$

It follows that $l \leqq \log_{1/\alpha}|V| + \log_{1/\alpha} 2 + 1$, and that the total cost of the tree is at most $C(\log_{1/\alpha}|V|+O(1))$. $\square$

In order to apply Lemma 3.1 to the algorithm of the preceding section, think of the algorithm as a rooted tree whose nodes correspond to recursive calls. The cost of a node is the time spent at this node. We will be able to argue that this cost is fading if we can show that the number of points decreases by a constant factor from one level of the recursion to the next.

LEMMA 3.2. *In Step 5 of the algorithm, each set S_i contains at most $\lceil 3|S|/4 \rceil$ points.*

Proof. Denote the four quadrants defined by l_1 and l_2 as NE, NW, SW, and SE. We claim that each set S_i contains points from at most three of the four quadrants. The assertion follows because each quadrant contains at least $\lfloor |S|/4 \rfloor$ of the points in S.

Note that no line parallel to l_1 intersects H_2' more than once, and that no line parallel to l_2 intersects H_1' more than once. Take a region R of the subdivision defined by H_1' and H_2'; its boundary can be decomposed into two connected chains, one in H_1' and one in H_2'. To show that R cannot intersect all four quadrants, we remove $a'b'c'$, and call the remaining branches of H_1' N (North) and S (South) and those of H_2' E (East) and W (West). Next, we connect each branch to point p by a straight line segment, as shown in Fig. 2.4. These modifications can only increase regions of the subdivision. Region R is bounded by only two of the four branches, say N and E, and cannot intersect SW because N intersects only NE and NW and E meets only NE and SE. □

Using the two lemmas we are ready to give the analysis of the algorithm still assuming general position of the points.

THEOREM 3.3. *The algorithm described in the previous section constructs the convex hull of a set S of n points in three-dimensional space in time $O(n \log^2 h)$ and storage $O(n)$ in the worst case, where $h = \text{ext}(S)$.*

Proof. Let $T(n, h)$ be the time-complexity of the algorithm, write S_i for the recursively considered subsets, and set $n_i = |S_i|$ and $h_i = |\text{ext}(S_i)|$. Then

$$T(n, h) = \begin{cases} O(n) & \text{if } h \leqq 3, \\ O(n \log h) + \sum_i T(n_i, h_i) & \text{otherwise.} \end{cases}$$

Think of $T(n, h)$ as a node in a tree, with cost $n \log h$ and children $T(n_i, h_i)$. Every time we recurse we find some new edges or facets on the convex hull, since otherwise there is no way to partition S into proper subsets which would contradict Lemma 3.2. So the number of nodes in the tree is at most the number of edges and facets on the hull which is $O(h)$.

Now, increase the cost of each node in the tree from $n_i \log h_i$ to $n_i \log h$. If c is the cost of a node then, by Lemma 3.2, the cost of each child is at most $\lceil 3c/4 \rceil$ and therefore the cost is fading.

At any one level of the tree, the algorithm works on different polygonal regions $R_1', R_2', \cdots, R_k'$ with boundaries $\mathcal{B}_1', \mathcal{B}_2', \cdots, \mathcal{B}_k'$. Let $|R_i'|$ be the number of points in R_i' (that is, inside and on $\mathcal{B}_i'$), and $|\mathcal{B}_i'|$ be the number of vertices of $\mathcal{B}_i'$. For the total cost at this level we have

$$\sum_{i=1}^{k} |R_i'| \log h \leqq \left(n + \sum_{i=1}^{k} |\mathcal{B}_i'| \right) \log h.$$

Since each $\mathcal{B}_i'$ is a simple polygon, each edge of the (planar) subdivision defined by

the $\mathscr{B}'_i$ occurs in at most two polygon boundaries on this level. Hence,

$$\sum_{i=1}^{k} |\mathscr{B}'_i| \leqq 6h - 12 \leqq 6n.$$

This implies that the cost at each level is bounded by $7n \log h$. Since the cost is fading, at each level it is bounded by $O(n \log h)$, and the total number of nodes in the tree is $O(h)$, we have $T(n, h) = O(n \log^2 h)$ by Lemma 3.1.

The $O(n)$ storage can be guaranteed if we declare H, $\mathscr{B}$, and S as global variables. For other subroutine calls, such as 4-partition, linear programming, two-dimensional convex hull and point location, $O(n)$ storage suffices and it is immediately returned after each call. □

Remark. To demonstrate that the above analysis is tight, we show that there are point sets for which the algorithm in § 2 indeed takes $\Theta(n \log^2 h)$ time. To see this consider the situation where the projections onto the XY-plane of extreme points form a $\sqrt{h}$-by-$\sqrt{h}$ grid and nonextreme points are evenly distributed in the grid. Since each call to the two-dimensional convex hull algorithm takes $\Theta(n \log h)$ time, and the depth of recursive calls is $\Theta(\log h)$, the algorithm described in § 2 runs in $\Theta(n \log^2 h)$ time.

4. Coping with degenerate point sets. As remarked earlier, point sets that are not in general position can be (conceptually) perturbed to satisfy the general position assumption with various definitions of this notion. For all details we refer to Edelsbrunner and Mücke (1990) where such a perturbation method is described. Still, there are two questions that need to be answered. First, how can we make sure that the perturbation does not change S in a way that significantly changes its convex hull and possibly increases the number of extreme points? Second, how does the perturbation affect the implementation of the necessary primitive operations?

We start with a brief review of the main features of the conceptual perturbation method, called SoS, of Edelsbrunner and Mücke (1990). SoS simulates an infinitesimal perturbation of the point coordinates that removes all degeneracies relevant to the algorithm of § 2. This is done by perturbing each coordinate differently, that is, there is a sequence of the coordinates so that the amount of perturbation of a coordinate is astronomically smaller than that of the preceding coordinates. Furthermore, because the perturbation is arbitrarily small everywhere, extreme points remain extreme and interior points remain interior. However, a point that is not extreme but lies on the boundary of the convex hull will end up either as an extreme point or in the interior of the convex hull. The decision made by SoS is based on the point and coordinate indices, arbitrarily assigned, and thus is by and large arbitrary.

If we accept this arbitrariness then the main theorem of this paper still applies, even if the point set S is not in general position. However, h must be redefined as the number of points that lie on the boundary of the convex hull, rather than the number of vertices of the convex hull. Another way to deal with nonextreme points on the boundary of the convex hull is to devise a sequence of the coordinates that guarantees that nonextreme points are perturbed below the upper hull. Such schemes are possible but tedious, and we refer to Rosenberger (1990, Chap. 5.2), where related ideas are explicated for the two-dimensional convex hull problem.

The next and related issue is how to simulate the perturbation. This is discussed at length in Edelsbrunner and Mücke (1990), except that they do not cover all primitive operations needed for our algorithm. The tricky part is in Steps 3 and 4 of our algorithm, where points are projected onto vertical planes Γ_1 and Γ_2. These planes as well as the directions of projection depend on the input points. As a consequence, the primitive

operations in the two-dimensional convex hull constructions are significantly more involved than in the plain plane case, although still of constant size. A detailed study of these operations together with possible simplifications is left as a future project.

5. Discussion. This paper presents an algorithm for constructing the convex hull of n points in three-dimensional space in worst-case time $O(n \log^2 h)$ and storage $O(n)$, where h is the number of vertices of the convex hull. It should be mentioned that the hidden constant in the big-O notation is rather large, although not astronomical. This is because the algorithms for the planar 4-partition, three-dimensional linear programming, and output-sensitive two-dimensional convex hull construction used in our algorithm all have large multiplicative constants.

It seems natural to ask if the time-complexity can be further reduced to $O(n \log h)$. The bottleneck of our method is the construction of two-dimensional hulls. All other operations can be done in time $O(n)$. It might also be interesting to see if the algorithm of this paper can be extended to four and higher dimensions; compare to the $O(n^2 + F \log h)$ algorithm of Seidel (1986).

Acknowledgments. The second author thanks Franco P. Preparata and Bernard Chazelle for discussions on the problem of this paper.

REFERENCES

M. BEN-OR (1983), *Lower bounds for algebraic computation trees*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 80-86.

D. R. CHAND AND S. S. KAPUR (1970), *An algorithm for convex polytopes*, J. Assoc. Comput. Mach., 17, pp. 78-86.

K. L. CLARKSON AND P. W. SHOR (1989), *Applications of random sampling in computational geometry*, II, Discrete Comput. Geom., 4, pp. 387-421.

R. COLE (1986), *Searching and storing similar lists*, J. Algorithms, 7, pp. 202-220.

M. E. DYER (1984), *Linear time algorithms for two- and three-variable linear programs*, SIAM J. Comput., 13, pp. 31-45.

H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI (1986), *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15, pp. 317-340.

H. EDELSBRUNNER (1987), *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, Germany, 1987.

H. EDELSBRUNNER AND E. P. MÜCKE (1990), *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, ACM Trans. Graphics, 9, pp. 66-104.

R. L. GRAHAM (1972), *An efficient algorithm for determining the convex hull of a finite planar set*, Inform. Process. Lett., 1, pp. 132-133.

J. E. HOPCROFT AND R. E. TARJAN (1974), *An efficient algorithm for graph planarity*, J. Assoc. Comput. Mach., 21, pp. 549-568.

D. G. KIRKPATRICK (1983), *Optimal search in planar subdivisions*, SIAM J. Comput., 12, pp. 28-35.

——— (1987), *Establishing order in planar subdivisions*, in Proc. 3rd Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, pp. 316-321.

D. G. KIRKPATRICK AND R. SEIDEL (1986), *The ultimate planar convex hull algorithm?*, SIAM J. Comput., 15, pp. 287-299.

N. MEGIDDO (1983), *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput., 12, pp. 759-776.

——— (1984), *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31, pp. 114-127.

——— (1985), *Partitioning with two lines in the plane*, J. Algorithms, 3, pp. 430-433.

F. P. PREPARATA AND S. J. HONG (1977), *Convex hulls of finite sets of points in two and three dimensions*, Comm. Assoc. Comput. Mach., 20, pp. 87-93.

F. P. PREPARATA AND M. I. SHAMOS (1985), *Computational Geometry—an Introduction*, Springer-Verlag, New York.

H. ROSENBERGER (1990), *Degeneracy control in geometric programs*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

N. SARNAK AND R. E. TARJAN (1986), *Planar point location using persistent trees*, Comm. Assoc. Comput. Mach., 29, pp. 669–679.

R. SEIDEL (1981), *A convex hull algorithm optimal for point sets in even dimensions*, Tech. Report 81-14, Department of Computer Science, University of British Columbia, British Columbia, Canada.

——— (1986), *Constructing higher-dimensional convex hulls at logarithmic cost per face*, in Proc. 18th Annual ACM Symposium on Theory Computing, Association for Computing Machinery, New York, pp. 404–413.

A. C. YAO (1981), *A lower bound to finding convex hulls*, J. Assoc. Comput. Mach., 28, pp. 780–789.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 270-277, April 1991

© 1991 Society for Industrial and Applied Mathematics

006

A GENERAL SEQUENTIAL TIME-SPACE TRADEOFF FOR FINDING UNIQUE ELEMENTS*

PAUL BEAME†

Abstract. An optimal $\Omega(n^2)$ lower bound is shown for the time-space product of any R-way branching program that determines those values which occur exactly once in a list of n integers in the range $[1, R]$ where $R \geqq n$. This $\Omega(n^2)$ tradeoff also applies to the sorting problem and thus improves the previous time-space tradeoffs for sorting. Because the R-way branching program is such a powerful model, these time-space product tradeoffs also apply to all models of sequential computation that have a fair measure of space such as off-line multitape Turing machines and off-line log-cost random access machines (RAMs).

Key words. lower bounds, time-space tradeoff, computational complexity, sorting, branching programs

AMS(MOS) subject classifications. 68P10, 68Q10, 68Q25

1. Introduction. The goal of producing nontrivial lower bounds on the time or space complexity for specific computational problems in $\mathcal{NP}$ has largely been elusive. Also, concentration on a single resource does not always accurately represent all of the issues involved in solving a problem. For some computational problems it is possible to obtain a whole spectrum of algorithms within which one can trade time performance for storage or vice versa. Thus the question of obtaining lower bounds that say something about time and space simultaneously has received considerable study as well.

The most interesting model for studying time-space tradeoff lower bounds that has been developed is the R-way branching program model. The R-way branching program is an unstructured model of computation that has unrestricted random access to its inputs and which makes no assumption about the way its internal storage is managed. The model is powerful enough that lower bounds proven in it apply to a wide variety of sequential computing models including off-line multitape Turing machines with random-access input heads. A particularly convenient model for which the lower bounds for R-way branching programs apply is that of a random access machine (RAM) with its input stored in a read-only memory, with a unit-cost measure of time and with its read-write storage charged on a log-cost basis.

The R-way branching program model was introduced by Borodin and Cook [BC82], who used it in showing the first nontrivial general sequential time-space tradeoff lower bound for any problem. They showed that any R-way branching program requires a time-space product of $\Omega(n^2/\log n)$ to sort n integers in the range $[1, n^2]$.

Since [BC82], time-space tradeoff lower bounds on R-way branching programs have been shown for a number of algebraic problems such as discrete Fourier transforms, matrix-vector products, and integer and matrix multiplication [Yes84], [Abr86]. In addition to these results, Reisch, in [RS82], has claimed an improvement of the sorting lower bound to $\Omega(n^2 \log\log n/\log n)$ using the same approach as in [BC82]. [RS82] presents an improvement of only one of the two key lemmas in [BC82]; this change appears to necessitate an overhaul of the second, more complex lemma as well in order to obtain the claimed bound. However, even this bound leaves a gap between the upper and lower bounds for sorting.

* Received by the editors March 16, 1989; accepted for publication (in revised form) June 28, 1990. This research was supported by National Science Foundation grant CCR-8858799.

† Computer Science Department, FR-35, University of Washington, Seattle, Washington 98195.

The first problem considered here, the UNIQUE ELEMENTS problem, is to produce, given an input list of n integers, a list of those integers that occur only once in the input. No particular order is required for the output of UNIQUE ELEMENTS. It is related to sorting, but its output provides much less information about the input than a sorted list provides. The main result of this paper is that any R-way branching program for UNIQUE ELEMENTS requires time T and space S such that $S \cdot T = \Omega(n^2)$ and that this bound is achievable using a simple RAM algorithm.

Borodin and Cook showed their time-space tradeoff for a somewhat unusual form of the sorting problem and derived bounds for the usual form of sorting by a straightforward reduction. In this paper their problem will be termed the RANKING problem and SORTING will be reserved for the usual form of output in which the elements are presented in sorted order. An easy reduction of UNIQUE ELEMENTS to the SORTING problem is shown which also yields an $\Omega(n^2)$ time-space product lower bound for SORTING. This improves the previous bounds in the amount of the tradeoff and also improves the range of inputs for which the bound holds. In this expanded range it is also shown that the tradeoff gap for sorting is closed, at least for R-way branching programs, since the $\Omega(n^2)$ time-space product is optimal.

A particularly remarkable feature of these results is the relative simplicity of the arguments required when compared with the involved arguments used in [BC82].

2. Definitions. An *$R(n)$-way branching program* consists of a directed acyclic rooted graph of out-degree $R = R(n)$ with each nonsink node labelled by an index from $\{1, \cdots, n\}$ and with the R out-edges of each node labelled $1, \cdots, R$. Edges of the branching program may also be labelled by a sequence of values from some output domain. The *size* of a branching program is the number of nodes it has. An R-way branching program is *levelled* if the nodes of the underlying graph are assigned levels so that the root has level 0 and the out-edges of a node at level l only go to nodes at level $l+1$.

Let $x = (x_1, \cdots, x_n)$ be an n-tuple of integers chosen from the range $[1, R]$. An R-way branching program computes a function of input x as follows. The computation starts at the root of the branching program. At each nonsink node v encountered, the computation follows the out-edge labelled with the value of x_i where i is the index that labels node v. (Variable x_i is *queried* at v.) The computation terminates when it reaches a sink node. The sequence of nodes and edges encountered is the *computation path* followed by x. The concatenation of the sequences of output values encountered along the path that x follows is the output of the branching program on input x.

The *time* used by a branching program is the length of the longest computation path followed by any input. The *space* used by a branching program is the logarithm base 2 of its size.

Any branching program can be levelled without changing its time and with at most squaring its size (see [BFK$^+$81]). Because this leaves the time used unchanged and changes the space used by no more than a factor of 2, it will usually be assumed without loss of generality that R-way branching programs are levelled.

Let $x = (x_1, \cdots, x_n)$ be an n-tuple of integers. An input value x_i is *unique* in x if there is no $j \neq i$ such that $x_i = x_j$. The UNIQUE ELEMENTS problem is, given an n-tuple of integers x as input, to produce as output a list (in arbitrary order) of exactly those values x_i that are unique in x.

In addition to the UNIQUE ELEMENTS problem the following two problems will also be of interest. The SORTING problem is, given an n-tuple of integers x as input, to produce as output the values of the x_i's in sorted (e.g., nondecreasing) order. The

RANKING problem is, given an n-tuple of integers x as input, to produce a list (in arbitrary order) of the ranks of all the inputs x_i in the sorted order of x where x_i's rank is output as a pair $(i, \text{rank}(x_i))$.

3. Unique elements.

THEOREM 1. *Any R-way branching program computing* UNIQUE ELEMENTS *for input integers in the range $[1, R]$, where $R \geqq n$, requires time T and space S such that $S \cdot T = \Omega(n^2)$.*

The general outline for the proof of this theorem is essentially the same as was used in the previous proofs of time-space tradeoff lower bounds for R-way branching programs [BC82], [Yes84], [Abr86], [Abr87] and which was originated in the context of comparison branching programs in [BFK+81]. The R-way branching program is broken up into layers and each layer is considered as a collection of shallow branching programs, one rooted at each node on the interlayer boundary. It is shown that any shallow branching program produces many output values for only a tiny fraction of the input n-tuples. Because the problem requires a large number of outputs to be made, if the time is not large then a large number of outputs must be made during some layer and therefore during some shallow branching program. The bound then follows since the total number of shallow branching programs must be sufficient to compensate for the small fraction of inputs for which each produces enough outputs.

Most problems for which time-space tradeoff lower bounds have been shown require a fixed large number of outputs, e.g., n outputs are required for sorting. In contrast, certain input vectors for UNIQUE ELEMENTS require few outputs or possibly none at all. However, a large number of outputs is required for a sufficiently large fraction of the inputs that the technique still applies.

It will be convenient to express the argument in a probabilistic format. Denote the uniform distribution on $[1, R]^n$ by U_R^n.

LEMMA 2. *If x is chosen at random from U_n^n then*

$$\Pr[x \textit{ contains } \geqq n/(2e) \textit{ unique elements}] > 1/(2e-1).$$

Proof. Let $u(x)$ denote the number of unique elements in x and let

$$Y_i = \begin{cases} 1 & \text{if } x_i \text{ is unique in } x, \\ 0 & \text{if not.} \end{cases}$$

Then $E[Y_i] = \Pr[Y_i = 1] = [(n-1)/n]^{n-1} = (1-1/n)^n/(1-1/n) > e^{-1}$. Thus

$$E[u(x)] = E\left[\sum_{i=1}^{n} Y_i\right] = \sum_{i=1}^{n} E[Y_i] > \frac{n}{e}.$$

Since there are never more than n unique elements in x, an application of Markov's inequality shows that $\Pr[u(x) \geqq n/(2e)] > 1/(2e-1)$. (For let $\alpha = \Pr[u(x) \geqq n/(2e)]$. Then $\alpha \cdot n + (1-\alpha) \cdot n/(2e) > E[u(x)] > n/e$. Solving for α yields the desired result.) □

For the UNIQUE ELEMENTS problem, say that an output value is *correct* for input x if it is the value of a unique element in x. Say that a branching program $\mathcal{P}$ *correctly outputs at least m values* on input x if all values output along the computation path in $\mathcal{P}$ that x follows are correct for x and at least m values are output along that computation path.

LEMMA 3. *Let $\mathcal{P}$ be an R-way branching program of height $\leqq n/4$ where $R \geqq n$. Let x be chosen at random from U_n^n. For $m \leqq n/4$,*

$$\Pr[\mathcal{P} \text{ correctly outputs at least } m \text{ values on input } x] \leqq e^{-m/2}.$$

Proof. Consider a computation path π in $\mathcal{P}$. Let Q_π be the set of indices of variables that are queried along π and V_π be the set of the first m values that are output along π. Some of the values in V_π can be values of variables queried along π but it is possible that some values in V_π are not. Call the values in V_π that are not values of any variable queried along π *extra* values and suppose that there are exactly k extra values in V_π. Let $s = n - |Q_\pi| - k$. Since $|Q_\pi| \leqq n/4$ and $k \leqq |V_\pi| = m \leqq n/4$, it follows that $s \geqq n/2$.

Assume that x has nonzero probability in U_n^n. The fact that an input x follows the path π in $\mathcal{P}$ only determines the values of the variables whose indices are in Q_π. The remaining $s+k$ variables are completely unconstrained so there are exactly n^{s+k} possible inputs in $[1, n]^n$ that can follow π in $\mathcal{P}$. For how many of these inputs are the values in V_π correct? In order for all the values in V_π to be correct it must be the case that, whatever the location of the k extra values in V_π among the $s+k$ unconstrained input variables, each of the remaining s variables must avoid all m values in V_π. Thus there are at most $(n-m)^s$ choices of the remaining s variables that would permit the values in V_π to be correct. Since there are exactly $(s+k)!/s!$ ways that the k extra values can occur in the input,

$$\begin{aligned}
\Pr[V_\pi \text{ is correct for } x \mid x \text{ follows } \pi] &\leqq \frac{(s+k)!}{s!} \cdot \frac{(n-m)^s}{n^{s+k}} \\
&< \left[\frac{s+k}{n}\right]^k \cdot \left[\frac{n-m}{n}\right]^s \\
&< \left[1 - \frac{m}{n}\right]^s \\
&\leqq \left[1 - \frac{m}{n}\right]^{n/2} \\
&< e^{-m/2}.
\end{aligned}$$

Since each input follows exactly one path π in $\mathcal{P}$, the statement of the lemma follows. □

Proof of Theorem 1. Consider an R-way branching program $\mathcal{B}$ for UNIQUE ELEMENTS. Assume without loss of generality that $\mathcal{B}$ is levelled. Suppose that $\mathcal{B}$ uses time T and space S, i.e., $\mathcal{B}$ has height T and has 2^S nodes. For convenience we can also assume without loss of generality that n is a multiple of 4 and that T is a multiple of $n/4$. (Since $\mathcal{B}$ must at least query all inputs to produce an output, T is at least n anyway.)

Divide the levels of $\mathcal{B}$ into layers of height $n/4$ where layer i consists of the portion of the branching program from level $(i-1)n/4$ to level $in/4$. View each node v at a level that is a multiple of $n/4$ as the root of an R-way subbranching program of height $n/4$ consisting of all nodes reachable from v in the layer whose levels start at v's level.

There are $T/(n/4) = 4T/n$ layers in $\mathcal{B}$. By Lemma 2, a large fraction of x in $[1, n]^n$ require at least $n/(2e)$ output values. For each such input x, at least $(n/(2e))/(4T/n) = n^2/(8eT)$ outputs must be made during some single layer. An input

reaches at most one node at each level and so reaches the root of only one subbranching program per layer. Now, for an input x chosen at random from U_n^n, by Lemma 3 each subbranching program can correctly output $\geqq n^2/(8eT)$ values on x only with probability $< e^{-(n^2/16eT)}$. (Note that $n^2/(8eT) < n/4$ since $T \geqq n$.)

Consider the probability, for x chosen at random from U_n^n, that there exists a subbranching program in $\mathcal{B}$ that correctly outputs at least $n^2/(8eT)$ values on input x. Since there are only 2^S nodes in $\mathcal{B}$, the number of subbranching programs that need to be considered is no more than 2^S and thus this probability is less than $2^S\, e^{-(n^2/16eT)}$. But, by Lemma 2, for x chosen at random from U_n^n the probability is $> 1/(2e-1)$ that the UNIQUE ELEMENTS problem requires at least $n/(2e)$ output values. Therefore $\mathcal{B}$ must have

$$2^S\, e^{-(n^2/16eT)} > \frac{1}{2e-1}$$

so that $S = \Omega(n^2/T)$, i.e., $ST = \Omega(n^2)$. □

Because the proof technique for Theorem 1 is probabilistic, it can be applied to show that the tradeoff for UNIQUE ELEMENTS holds for *average* time and space as well (see [Abr86]) in the case that the input integers are chosen uniformly from $[1, n]$.

Any problem for which n input variables must be read before some output is produced requires branching program time $T \geqq n$ and therefore space $S \geqq \log n$. Thus, for inputs in the range $[1, n]$, the following theorem demonstrates that the tradeoff in Theorem 1 is optimal.

THEOREM 4. *For any S with $n \geqq S \geqq \log n$ there is an n-way branching program that solves the* UNIQUE ELEMENTS *problem for inputs in the range $[1, n]$ using $O(S)$ space and $O(n^2/S)$ time.*

Proof. The n-way branching program is a straightforward implementation of the following RAM program:

```
ALGORITHM UNIQUE ELEMENTS.
b ← 0
for j = 1 to ⌈n/S⌉ do
   for i = 1 to S do
        A[i] ← 0
     end for
   for i = 1 to n do
        if b < x_i ≦ b + S then do
             k ← x_i − b
             if A[k] < 2 then A[k] ← A[k] + 1
          end if
     end for
   for i = 1 to S do
        if A[i] = 1 then Output b + i
     end for
   b ← b + S
end for
```

Each of the S entries in the array A only contains either 0, 1, or 2, and the other variables only store values that require only $O(\log n)$ bits of storage. Each of the $O(n/S)$ passes through the outer loop uses only $O(n)$ time. Thus the program uses $O(S)$ space and $O(n^2/S)$ time. □

The technique of Theorem 4 does not apply if the range of inputs is significantly larger, say $[1, n^c]$ for $c > 1$. For inputs in this range the best upper bound known is an $O(n^2 \log n)$ time-space product used by a number of straightforward algorithms. If this really is the best possible then it leaves a $\log n$ gap which seems to be difficult to close using the approach of Theorem 1 since larger ranges of inputs only increase the likelihood that inputs are unique. It would be interesting to close this gap.

4. Sorting. Borodin and Cook's time-space tradeoff [BC82] of $S \cdot T = \Omega(n^2/\log n)$ for sorting n distinct integers actually holds for the RANKING problem on inputs in the range $[1, n^2]$. In order to get the bound for SORTING they use an easy reduction which uses small amounts of additional time and space from RANKING for inputs in the range $[1, n^2]$ to SORTING for inputs in the range $[1, n^3]$. In [RS82], Reisch's better bound of $S \cdot T = \Omega(n^2 \log \log n/\log n)$ is for RANKING distinct integers in the range $[1, n \log n/\log \log n]$ and gives a similar reduction in the range for SORTING. The above bounds for UNIQUE ELEMENTS will yield an improvement in both of these results.

With the output of the SORTING problem on inputs in the range $[1, n]$ it is possible to solve the UNIQUE ELEMENTS problem for inputs in the range $[1, n]$ using only small amounts of additional storage and time. Intuitively think of the index and value of the most recently generated output of the sorting program being stored along with a flag bit that is 1 if the stored value is the only one of its kind seen so far. When a new output for the sorting problem is produced it is compared with the stored value. If the flag bit is 1 and the compared values are different, then the stored value is output as a unique element. The flag and the stored value are then reset appropriately.

In the context of n-way branching programs the reduction is implemented by creating $2n$ copies of the program for SORTING to handle the different values of the stored input as well as the flag bit. The test of the flag bit and of the stored output of the sorting problem against the new one is handled implicitly since the state information of the modified branching program will be sufficient. The edges on which outputs occurred in the SORTING program have to be routed to the appropriate copy of the original program and the outputs for SORTING have to be replaced by the unique elements also where appropriate. This reduction uses no additional time and only $O(\log n)$ additional space. Thus the following corollary of Theorem 1 is obtained.

COROLLARY 5. *Any R-way branching program for* SORTING *for input integers in the range $[1, R]$, where $R \geqq n$, requires time T and space S such that $S \cdot T = \Omega(n^2)$.*

It is worth remarking that the bounds in [BC82] and [RS82] hold for *distinct* inputs to the SORTING problem. However, if the range of inputs is restricted to $[1, n]$, as could be the case for the bound proven here, then the problems for distinct inputs are trivially solvable. The statement to Corollary 5 does hold for distinct inputs except that R must be at least n^2 in this case. The reduction from UNIQUE ELEMENTS to sorting distinct values is achieved by appending an input's index to its value as the low order $\log n$ bits. The sorting algorithm is used as above except that outputs are not checked for simple uniqueness but rather for uniqueness in an interval of n consecutive values in $[1, R]$. In fact, in general, SORTING can be used in this same way to solve the following UNIQUE INTERVALS problem: Given an n-tuple of integers x in the range $[1, R]$, produce a list (in arbitrary order) of those $i \in [1, n]$ such that the interval $[(i-1)R'+1, iR']$ for $R' = \lceil R/n \rceil$ contains a unique value from x.

The average case argument for UNIQUE ELEMENTS with input integers chosen uniformly from $[1/n]$ that was alluded to in the last section can be used to show that the tradeoff for SORTING in Corollary 5 also holds for average time and space for input integers chosen uniformly from $[1, R]$ for $R \geqq n$: The key observation is that the

probabilistic analysis of a branching program for the UNIQUE INTERVALS problem for a random input n-tuple in $[1, R]$ is essentially the same as for UNIQUE ELEMENTS in the range $[1, n]$. Then the reduction from the UNIQUE INTERVALS problem to SORTING gives the desired average case lower bound tradeoff.

The following theorem shows that Corollary 5 for SORTING is optimal for input integers in the range $[1, n]$ although the branching program that shows this is not obviously expressible as a RAM program, as was the case with the program for UNIQUE ELEMENTS in the proof of Theorem 4. As was the case for UNIQUE ELEMENTS, there is a $\log n$ gap for inputs in ranges such as $[1, n^2]$.

THEOREM 6. *For any S with $n \geqq S \geqq \log n$ there is an n-way branching program that solves the* SORTING *problem for inputs in the range $[1, n]$ using $O(S)$ space and $O(n^2/S)$ time.*

Proof. First consider the situation when $S = n$. The output of the SORTING problem only depends on the number of inputs of each value, not on their order in the input. There are $\binom{n+k-1}{n-1}$ ways of selecting k numbers in the range $[1, n]$. The branching program has a root node and a node for each of the ways of selecting k inputs from $[1, n]$ where $1 \leqq k \leqq n$. The program makes one pass through the inputs, traversing the nodes in the obvious way. On the edges leading to each node that describes the entire selection of n numbers, the entire sorted sequence is output. Since $\binom{n+k-1}{n-1} < 2^{n+k-1}$ the program uses $O(n)$ space and $O(n)$ time.

The modification of the branching program for smaller values of S gets a little trickier. It would be nice simply to make a pass through the inputs and record how many inputs there are of each value in the range 1 through S, then output that prefix of the sorted list and proceed on through the ranges $S+1$ to $2S$, $2S+1$ to $3S$, etc. The problem is that there might be too many inputs in some of these ranges. In order to handle this, the program never stores more than S values below the largest value in the range for which it is currently recording. When the $(S+1)$st value is found, the upper end of the range being recorded is lowered until the count of inputs below the largest value in the range is at most S. At the end of the pass through the input the portion of the sorted list corresponding to the range is output. Each range is initially set at length S and starts where the previous range left off.

To store the necessary markers and to record the number of inputs which take on the largest value of the range requires only $O(\log n)$ space and the record for the remainder of the range requires at most $O(S)$ space. Each pass through the input either reduces the number of inputs by S or outputs the sorted list for all values in a range of length S. Thus there are a total of $O(n/S)$ passes through the input for a total time of $O(n^2/S)$. □

5. Conclusions. This paper shows the first optimal time-space tradeoff lower bounds for UNIQUE ELEMENTS and for SORTING that apply to any general sequential model that has a fair measure of space, without restrictions on its mode of accessing the inputs or on the structure of its computation. In comparison with previous work for SORTING, the bounds are better and the argument is considerably simpler. In addition, the UNIQUE ELEMENTS problem deals with questions of distinctness in a direct way and arguments about it may provide intuition that will be helpful for studying the element distinctness problem which seems to be the next natural problem to be attacked in the area of time-space tradeoffs for R-way branching programs.

There is a sense in which the lower bounds proven here for SORTING and those in [BC82] and [RS82] are orthogonal. The bounds in [BC82] and [RS82] hold for RANKING as well as for the SORTING problem. There do not seem to be any reductions

using small time and space between the RANKING and UNIQUE ELEMENTS problems, so the previous bounds for RANKING are not improved by these results.

Clearly, the most interesting open problem in the area of time-space tradeoff is that posed in [BC82]—namely, to prove a nontrivial tradeoff for some decision problem in an unrestricted unstructured model like the R-way branching program. As mentioned above, the element distinctness problem seems to be a natural candidate. Good time-space tradeoff lower bounds have been shown for models with unstructured computation but restricted access to the inputs [Kar86], as well as for structured models with unrestricted access to the inputs [BFM+87], [Kar86], [Yao88]. However, there appears to be a big stumbling block in the way of achieving similar results for R-way branching programs since, so far, the limit of time-space product lower bounds for R-way branching programs has been $O(nm)$ where n is the number of inputs and m is the number of outputs.

In one aspect, the inability to prove tradeoffs for decision problems may be due to a lack of intuition about measures of "progress" for solving them; in structured models a good measure of progress for the problem of element distinctness has led to interesting and nearly optimal time-space tradeoff lower bounds [BFM+87], [Yao88]. However, it also seems likely that in addition to better measures of progress a more sophisticated handling of how branching programs make their progress is also needed. In the general framework for the time-space tradeoff shown here and in virtually all similar tradeoffs for branching programs it is granted that every input that could make good use of a subbranching program has been routed to the root of that subbranching program. The recent result of Yao [Yao88] for comparison branching programs uses a more careful accounting but it remains to be seen if even this accounting will be effective for R-way branching programs.

Acknowledgments. I would like to thank Richard Anderson, Larry Ruzzo, Martin Tompa, All Borodin, and Steve Cook for several helpful discussions and suggestions concerning these results.

REFERENCES

[Abr86] K. ABRAHAMSON, *Time-space tradeoffs for branching programs contrasted with those for straight-line programs*, in Proc. 27th IEEE Symposium on the Foundations of Computer Science, 1986, Toronto, Ontario, Canada, IEEE Computer Society, Washington, DC, pp. 402-409.

[Abr87] ———, *Generalized string matching*, SIAM J. Comput., 16 (1987), pp. 1039-1051.

[BC82] 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.

[BFK+81] A. BORODIN, M. FISCHER, D. KIRKPATRICK, N. LYNCH, AND M. TOMPA, *A time-space tradeoff for sorting on non-oblivious machines*, J. Comput. System Sci., 22 (1981), pp. 351-364.

[BFM+87] 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.

[Kar86] M. KARCHMER, *Two time-space tradeoffs for element distinctness*, Theoret. Comput. Sci., 47 (1986), pp. 237-246.

[RS82] S. REISCH AND G. SCHNITGER, *Three applications of Kolmogorov complexity*, in Proc. 23rd IEEE Symposium on the Foundations of Computer Science, 1982, Chicago, IL, IEEE Computer Society, Washington, DC, pp. 45-52.

[Yao88] A. YAO, *Near optimal time-space tradeoff for element distinctness*, in Proc. 29th IEEE Symposium on the Foundations of Computer Science, 1988, White Plains, NY, IEEE Computer Society, Washington, DC, pp. 91-97.

[Yes84] 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. 20, No. 2, pp. 278-290, April 1991
© 1991 Society for Industrial and Applied Mathematics
007

THE POWER OF ALTERNATING ONE-REVERSAL COUNTERS AND STACKS*

OSCAR H. IBARRA† AND TAO JIANG‡

Abstract. The relation between reversals and alternation is studied in two simple models of computation: the 2-counter machine with a one-way input tape whose counters make only one reversal (1-reversal 2CM) and the one-way pushdown automaton whose pushdown store makes only one reversal (1-reversal PDA). The following is shown: (a) alternating 1-reversal 2CM's accept all recursively enumerable languages; (b) alternating 1-reversal PDA's accept exactly the languages accepted by exponential time-bounded deterministic TM's. The first improves on the known result that alternating 1-reversal 4CM's accept all recursively enumerable languages. The second improves an earlier result that alternating PDA's with no restrictions on reversals accept exactly the exponential-time languages.

Key words. alternation, reversal, counter machine, pushdown automata, Turing machine, computational complexity, recursively enumerable set

AMS(MOS) subject classifications. 68Q05, 68Q15, 68Q45

1. Introduction. A well-known result in automata theory is the equivalence of Turing machines (TM's) and two-counter machines: every recursively enumerable language can be accepted by a two-counter machine with a one-way input tape (2CM for short) [HOPC79]. When the counters are restricted to make only one reversal (i.e., once a counter decreases its count, it can no longer increase), the power of the device is substantially reduced. Call this restricted device 1-reversal 2CM (or 1-reversal mCM when there are m counters, $m>0$). One-reversal mCM's have been studied in many places in the literature (see, e.g., [BAKE74], [CHAN81a], [GURA81], [IBAR78]). For example, it was shown in [IBAR78] that the class of languages accepted by 1-reversal mCM's has decidable properties similar to those of the regular sets. Languages accepted by nondeterministic 1-reversal mCM's have semilinear Parikh maps [IBAR78] and can be accepted by nondeterministic $\log n$ space-bounded TM's [BAKE74], [GURA81].

Generalizing nondeterminism to alternation can significantly increase the computing power of 1-reversal mCM's. For example, it was shown in [HROM85] that P (equal to the class of languages accepted by deterministic TM's in polynomial time) equals the class of languages accepted by alternating 1-reversal mCM's in polynomial time and that every recursively enumerable (r.e.) language can be accepted by an alternating 1-reversal 4CM. We describe the idea behind the proof of the latter result. Let M_1 be an arbitrary 2CM. An alternating 1-reversal 4CM M_2 simulates M_1 as follows. Each counter C of M_1 is simulated by two counters C_1 and C_2 of M_2. The value of C is represented by the difference of the values of C_1 and C_2. Incrementing (decrementing) C corresponds to incrementing C_1 (C_2). Zero-testing of C corresponds to checking that the values of C_1 and C_2 are equal.

* Received by the editors January 1, 1990; accepted for publication (in revised form) June 25, 1990.

† Department of Computer Science, University of California, Santa Barbara, California 93106. This author's research was supported in part by National Science Foundation grants DCR-8604603 and CCR-8918409.

‡ Department of Computer Science and Systems, McMaster University, Hamilton, Ontario L8S 4K1, Canada. This author's research was supported in part by National Science Foundation grants DCR-8420935 and DCR-8604603, and Natural Sciences and Engineering Research Council of Canada operating grant OGP 0046613.

By slightly modifying the construction above, one can reduce the number of counters of M_2 to 3. In this paper, we show that, in fact, every r.e. language can be accepted by an alternating 1-reversal 2CM. The proof uses an entirely new technique which is much more involved. We note in passing two related results: every r.e. language can be accepted by a nondeterministic one-way two-pushdown machine whose pushdown stores make only one reversal [BAKE74]; every r.e. language can be accepted by an alternating 1-tape 1-counter machine (i.e., a single-tape TM with one counter) which runs in constant reversals [YAMA87]. Our result cannot be improved in the sense that every language accepted by an alternating 1CM (whose counter is unrestricted) is in $\bigcup_{c>0}$ DTIME(c^n). Alternating 1CM's are quite powerful, even when the counter is restricted to make at most one reversal. For example, we show that Dyck languages on k letters [HOPC79], parenthesis languages [LYNC77], and linear context-free languages can be accepted by alternating 1-reversal 1CM's.

Next consider the one-way pushdown automaton (PDA). Define a 1-reversal PDA to be one whose pushdown store makes only one reversal (i.e., once the pushdown store pops it can no longer push). It is well known that nondeterministic PDA's accept exactly the context-free languages (CFL's) and that nondeterministic 1-reversal PDA's accept exactly the linear CFL's [HOPC79], the latter being a proper subset of the CFL's. (Linear CFL's can easily be accepted by nondeterministic log n space-bounded TM's.) When we make the devices alternating, we show that alternating PDA's are equivalent to alternating 1-reversal PDA's. It is known that alternating PDA's accept exactly the languages in $\bigcup_{c>0}$ DTIME(c^n) [CHAN81b], [LADN84]. This can now be improved: alternating 1-reversal PDA's accept exactly the languages in $\bigcup_{c>0}$ DTIME(c^n).

This paper only studies 1-reversal machines. Bounding the number of reversals by a small nonconstant function of the input length, for TM's, PDA's, and CM's, has also been studied before [BAKE74], [CHAN81a], [FISH68], [HART68], [KAME70], [RYTT85]. There is renewed interest in reversal complexity because of its connection to uniform Boolean circuit depth and parallel time (see, e.g., [CHEN87], [HONG80], [PIPP79]).

2. Alternating 1-reversal 2-counter machines. An m-counter machine (mCM for short) is a one-way finite automaton augmented with m counters, $m>0$. A k-reversal mCM is an mCM with the property that each counter reverses (i.e., changes from increasing to decreasing mode and vice versa) at most k times in any computation. An alternating (k-reversal) mCM is a straightforward generalization of a nondeterministic (k-reversal) mCM, by allowing the machine to make not only existential moves but also universal moves. It is known that languages accepted by nondeterministic k-reversal mCM's are in NSPACE(log n) [BAKE74], [GURA81]. Here we show that alternating 1-reversal 2CM's accept all r.e. languages. This shows that alternation drastically increases the power of 1-reversal 2CM's. It is well known that deterministic 2CM's (with unrestricted counters) accept all r.e. languages. Hence the above result shows a trade-off between two resources: reversals and alternation.

THEOREM 1. *Every* r.e. *language can be accepted by an alternating* 1*-reversal* 2CM.

The proof consists of first showing that every r.e. language can be accepted by a deterministic 2CM which increases and decreases its counters in a very special way, and then showing that this machine can be simulated by an alternating 1-reversal 2CM. For simplicity, we will only consider binary languages (i.e., subsets of $\{0, 1\}^*$). We assume that the one-way input has a right endmarker, \$. We also assume, without loss

of generality, that a counter machine accepts an input by entering an accepting state with the counters reset to 0.

A configuration of an alternating 2CM is a 5-tuple (q, x, i, n_1, n_2), where
- q is the current state,
- x is the input,
- i is the distance between the input head and the right endmarker \$,
- $n_1, n_2 \geqq 0$ are numbers stored in the first counter and the second counter, respectively.

Throughout, q_0 denotes the initial state of a machine.

DEFINITION. Let N be the set of nonnegative integers. A relation $R \subseteq \{0, 1\}^* \times N^3$ is *verified* by an alternating 2CM *in k reversals*, where $k \geqq 0$, if for every $(x, i, n_1, n_2) \in \{0, 1\}^* \times N^3$, when (q_0, x, i, n_1, n_2) is given as the initial configuration, the machine accepts if and only if $(x, i, n_1, n_2) \in R$ and the machine makes at most k reversals.

Obviously, if a relation R is verified by an alternating 2CM in k reversals, then the relation $R^T = \{(x, i, n_1, n_2) \mid (x, i, n_2, n_1) \in R\}$ is also verified by an alternating 2CM in k reversals. We are only interested in the relations that can be verified by alternating 2CM's in zero or one reversals. Note that, if an alternating 2CM verifies some relation in zero reversals, it can only decrease its counters when started on the initial configuration.

LEMMA 1. *The relation* $R_1 = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, 0 \leqq i \leqq |x|$, *and* $0 < n_2 = 2n_1\}$ *and* $R_2 = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, 0 \leqq i \leqq |x|$, *and* $0 < n_2 \leqq 2n_1\}$ *are verified by alternating* 2CM's *in zero reversals.*

Proof. The proof is obvious. Even a deterministic 0-reversal 2CM can perform the verifications. □

The next lemma says that an alternating 2CM can check, in zero reversals, whether the number stored in one of its counters is a power of two.

LEMMA 2. $R_3 = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, 0 \leqq i \leqq |x|, 0 < n_2 \leqq 2n_1$, *and* $n_2 = 2^m$ *for some* $m \geqq 0\}$ *and* $R_3' = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, 0 \leqq i \leqq |x|, 0 < n_2 \leqq 2n_1$, *and* $n_2 \neq 2^m$ *for any* $m \geqq 0\}$ *are verified by alternating* 2CM's *in zero reversals.*

Proof. We show how R_3 is verified by an alternating 2CM in zero reversals. The same idea also applies to R_3'.

Given (q_0, x, i, n_1, n_2) as the initial configuration, the machine first generates a side process to verify that $0 < n_2 \leqq 2n_1$, using the machine for R_2. Then it guesses whether $n_2 = 1$. If the machine guesses that $n_2 = 1$, it verifies that the guess is correct and accepts. Otherwise, it decrements the first counter by some $d_1 \geqq 0$ and verifies that $n_2 = 2(n_1 - d_1)$, using the machine for R_1. Then it decrements the second counter by some $d_2 > 0$ and verifies that $n_1 - d_1 = 2(n_2 - d_2)$, and so on. The machine keeps "halving" its two counters alternately until it (nondeterministically) chooses to stop. Then it verifies that one counter contains 1 and the other contains 2. □

DEFINITION. Let $n > 0$ be the number contained in a counter (of some alternating 2CM). Then there is a unique pair of nonnegative integers m and d such that $d < 2^m$ and $n = 2^m + d$. We call 2^m and d the *base* and *offset* of the counter, respectively.

Lemma 2 gives a way to calculate the base of a counter in zero reversals. The following lemma shows that an alternating 1-reversal 2CM can increase the base of a counter without changing its offset.

LEMMA 3. $R_4 = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, 0 \leqq i \leqq |x|$, *and* $n_1 = 2^m + d$, $n_2 = 2^{m+1} + d$ *for some* $m \geqq 0$ *and* $0 \leqq d < 2^m\}$ *is verified by an alternating* 2CM *in zero reversals.*

Proof. The basic idea is as follows. Given an initial configuration (q_0, x, i, n_1, n_2), the machine verifying R_4 guesses and verifies that one of the following is true: (1) Both counters contain powers of 2, or (2) Neither counter contains a power of 2. In case (1), it verifies that $n_2 = 2n_1$, using the machine for R_1. In case (2), it repeatedly decreases both counters until their values become powers of 2. Then it verifies that the value of the second counter is twice the value of the first counter. The details of the construction are given below.

```
procedure VR4(q0, x, i, n1, n2);
  loop
    do existentially
      (i) do universally
            (a) Verify 0 < n2 = 2n1 and both n1 and n2 are powers of 2;
            (b) exit;
          end;
     (ii) do universally
            (a) Verify 0 < n1 ≦ 2n2 and n1 is not a power of 2;
            (b) Verify 0 < n2 ≦ 2n1 and n2 is not a power of 2;
            (c) n1 := n1 − 1; n2 := n2 − 1;
          end
    end
  endloop
end. {The end of VR4}
```
□

Lemma 4 shows that an alternating 1-reversal 2CM can compute exponential functions.

LEMMA 4. *$R_5 = \{(x, i, n_1, n_2) | x \in \{0, 1\}^*, n_1 \geqq 0$, and $n_2 = 2^{n_1}\}$ is verified by an alternating* 2CM *in one reversal.*

Proof. The machine verifying R_5 works as follows. Given the initial configuration (q_0, x, i, n_1, n_2), the machine first verifies that n_2 is a power of 2 using the machine for R_3. Then it decreases the first counter by 1 and divides the second counter by 2. To do the division, the machine repeatedly decreases the second counter by 1 until it guesses that the counter value has reached the next power of 2. To make sure that the guess is correct, the machine generates side processes to verify that each intermediate value of the second counter is not a power of 2, using the machine for R_3', and the last value of the second counter is a power of 2, using the machine for R_3. Then the machine decreases the first counter by 1 and divides the second counter by 2, and so on. This process is continued until the first counter contains 0 and the second counter contains 1 simultaneously.

The following procedure describes the detailed construction.

```
procedure VR5(q0, x, i, n1, n2);
  do existentially
    I. Verify n1 = 0 and n2 = 1;
   II. do universally
         (i) Verify n2 is a power of 2;
        (ii) {First n2 ← n2/2, n1 ← n1 + n2/2}
             n2 := n2 − 1; n1 := n1 + 1;
             loop
               do existentially
                 (a) do universally
                       (a1) Verify 0 ≦ n2 ≦ 2n1 and n2 is a power of 2;
```

```
                (a2) exit;
              end;
          (b) do universally
                (b1) Verify n2 is not a power of 2;
                (b2) n2 := n2 - 1; n1 := n1 + 1;
              end
        end
      endloop;
      {Now verify n2 = 2^m and n1 = 2^m + m + 1 for some m ≧ 0}
      loop
        do existentially
          (a) Verify n1 = 2 and n2 = 1;
          (b) {n2 ← n2/2, n1 ← n1 - n2/2 - 1}
              n1 := n1 - 2; n2 := n2 - 1;
              loop
                do existentially
                  (b1) do universally
                         (b11) Verify 0 ≦ n2 ≦ 2n1 and n2
                               is a power of 2;
                         (b12) exit;
                       end;
                  (b2) do universally
                         (b21) Verify 0 ≦ n2 ≦ 2n1 and n2
                               is not a power of 2;
                         (b22) n2 := n2 - 1; n1 := n1 - 1;
                       end
                end
              endloop
        end
      endloop
    end
  end
end. {The end of VR5.}
```

□

DEFINITION. Define the function num : $\{0, 1\}^* \to N$ as follows: $\text{num}(a_1 a_2 \cdots a_n) = 2^n + a_1 2^{n-1} + a_2 2^{n-2} + \cdots + a_n$. For each string $x \in \{0, 1\}^*$, $\text{num}(x)$ is called the *corresponding number* of x.

It is easy to see that the function num is one-to-one. The next lemma says that an alternating 1-reversal 2CM can compute the corresponding number of an input and store it in one of its counters.

LEMMA 5. $R_6 = \{(x, i, n_1, n_2) \mid x = a_1 a_2 \cdots a_n \in \{0, 1\}^*, 0 \leqq i \leqq n, n_1 = 2^i, n_2 = \text{num}(a_{n+1-i} \cdots a_n)\}$ *is verified by an alternating* 2CM *in zero reversals.* (*Note that for a* $(x, i, n_1, n_2) \in R_6$, *if* $i = |x|$, *then* $n_2 = \text{num}(x)$.)

Proof (sketch). The alternating 2CM verifying R_6 works as follows. Given an initial configuration (q_0, x, i, n_1, n_2), first it verifies that $0 < n_1 \leqq n_2 < 2n_1$ and n_1 is a power of 2. Then it decrements the first counter by some $d > 0$ (d is supposed to be $n_1/2$) and, simultaneously, decrements the second counter by d if the current input symbol is 0, or by $2d$ if the current input symbol is 1. Meanwhile, it verifies that for each d_1, $0 < d_1 < d$, $n_1 - d_1$ is not a power of 2. Then the machine shifts the input head one square to the right. Let n_1' and n_2' be the new contents of the counters. The machine

repeats the above procedure for the configuration $(q_0, x, i-1, n_1', n_2')$. This repetition is continued until the input head reaches the right endmarker \$. At that moment, it verifies that both counters contain 1's. □

Before giving the proof of Theorem 1, we recall how a deterministic 1-tape TM can be simulated by a deterministic 2CM [HOPC79]. Let M be a deterministic 1-tape TM and M' be the simulating deterministic 2CM. Given input x, the computation of M' consists of two phases. In the first phase, M' computes $2^{\mathrm{num}(x)}$, stores it in one counter, and resets the other counter to zero. In the second phase, M' simulates the changes of M's configuration by multiplying/dividing the numbers in its counters by 2, 3, 5, or 7. Thus the second phase can be divided into a sequence of subphases such that in each subphase, M' decreases one counter from a positive number d_1 to 0 and increases the other counter from 0 to some positive number d_2. For the details of the simulation, see [HOPC79]. Note that the input head of M' is not used in the second phase.

Now we are ready to prove Theorem 1.

Proof of Theorem 1. We show that a deterministic 1-tape TM can be simulated by an alternating 1-reversal 2CM. Let M be a deterministic 1-tape TM and M_1 be a deterministic 2CM which simulates M in the way described in the last paragraph above. We construct an alternating 1-reversal 2CM M_2 to simulate M_1. From the above discussion, it suffices to show that M_2, when given an input x, can simulate the two phases of the computation of M_1 on x.

Let C_1 and C_2 be the two counters of M_2. The computation of M_2 (on x) is divided into three stages. In the first stage, M_2 computes $2^{\mathrm{num}(x)}$ and stores it in C_2 as follows. Initially both C_1 and C_2 contain zeros. M_2 increments C_1 by n_1 and C_2 by n_2 for some $n_1, n_2 > 0$, and verifies that $n_1 = 2^{|x|}$ and $n_2 = \mathrm{num}(x)$ using the machine for R_6. Then it increments C_1 by d_1 for some $d_1 > 0$ and verifies that $n_1 + d_1 = 2^{\mathrm{num}(x)}$ using the machine for R_5^T.

In the second stage, M_2 increments C_2 by d_2 for some $d_2 > 0$ and verifies that $3(n_1 + d_1) = n_2 + d_2$. Then it increments C_1 by d_3 for some $d_3 > 0$ and verifies that $n_1 + d_1 + d_3 = n_2 + d_2$. (Now the base of C_1 = the base of $C_2 = 2^{\mathrm{num}(x)+1}$ and the offset of C_1 = the offset of $C_2 = 2^{\mathrm{num}(x)}$.)

In the third stage, M_2 simulates the second phase of M_1's computation by going through the subphases one by one. We describe how M_2 simulates a subphase SP. Suppose that in SP M_1 changes its state from q_1 to q_2, decreases one of its counters from d_1 to 0, and increases the other from 0 to d_2. Also suppose that at the end of the simulation of the subphase preceding SP, q_1 is remembered in M_2's finite control, the offset of $C_1 = d_1$, the offset of $C_2 = d_0$, and the base of C_1 = the base of $C_2 = 2^m$. (The reader can verify that these conditions are satisfied at the end of the second stage.)

M_2 increments C_2 by e_1 for some $e_1 > 0$ and verifies that $2^m + d_0 + e_1 = 2^{m+1} + d_1$, using the machine for R_4. It repeats this "base-raising" process 3 times. (Now C_1 contains $2^{m+2} + d_1$ and C_2 contains $2^{m+3} + d_1$.) Then it increments C_1 by some $e_2 > 0$ and verifies that the current base of C_1 is 2^{m+3}. Let $d_2' = d_1 + e_2 - 2^{m+2}$ (i.e., d_2' is the current offset of C_1). M_2 guesses a state (of M_1) q_2' and remembers it in the finite control. Then it verifies that $d_2' = d_2$ and $q_2' = q_2$, and, at the same time, initiates the simulation of the subphase following SP.

The verifications of $d_2' = d_2$ and $q_2' = q_2$ are done by directly simulating the operations of M_1 in SP. M_2 uses C_1 to imitate the decreasing counter of M_1 and C_2 to "oppositely" imitate the increasing counter, i.e., when the decreasing counter of M_1 is decremented by 1, C_1 is decremented by 1, and when the increasing counter of M_1 is incremented by 1, C_2 is decremented by 1. It verifies that the offsets of C_1 and C_2

become zero at the same time, and the state of M_1 is then q'_2.

It is easy to see that M_2 makes at most one reversal on all inputs. □

Since alternating 2CM's accept only r.e. languages, we have the following corollary.

COROLLARY 1. *Alternating* 1*-reversal* 2CM's *are equivalent to alternating* 2CM's.

3. Alternating 1-reversal 1-counter machines. Every language accepted by an alternating 1CM is in $\bigcup_{c>0}$ DTIME (c^n) [LADN84]. Although we do not know the exact complexity of alternating 1CM's, we can show that they are quite powerful even when the counter is restricted to make at most one reversal. We show, e.g., that Dyck languages on k letters [HOPC79], parenthesis languages [LYNC77], and linear CFL's can be accepted by alternating 1-reversal 1CM's.

A Dyck language on k letters ($k \geqq 1$) [HOPC79] is a CFL defined by the grammar $G_k = \langle \{S\}, \Sigma_k, P, S \rangle$, where $\Sigma_k = \{[_1, [_2, \cdots, [_k,]_1,]_2, \cdots,]_k\}$ and $P = \{S \to [_i S]_i, S \to SS, S \to [_i\,]_i \mid 1 \leqq i \leqq k\}$. The Dyck language on k letters is denoted by D_k. The significance of Dyck languages arises from the fact that they form an example of a "hardest CFL" [GREI73] using homomorphic reductions. It is known that D_k is in *ATIME* (log n) [IBAR88]. We will show that D_k can be recognized by an alternating 1-reversal 1CM. The proof of this result is based on a nice characterization of Dyck languages given in the next lemma. The characterization is a variation of the one given in [IBAR88] and is more suitable for implementation on an alternating 1-reversal 1CM. There is a simple characterization of D_1 [HOPC79]: a string is in D_1 if and only if x has an equal number of $[_1$'s and $]_1$'s, and in any prefix of x, the number of $[_1$'s is greater than or equal to the number of $]_1$'s. We call a string that satisfies this condition *balanced.*

For simplicity, we consider only the language D_2. The generalization to arbitrary k can be carried out in an obvious manner.

DEFINITION. Let w be a string in D_1. We say that w is *reducible* if there are x and y both nonnull, such that $w = xy$ and $x, y \in D_1$. If w is not reducible, it is called *irreducible.*

DEFINITION. For a string w, let $w_{i:j}$ denote the substring starting at the ith position and ending at the jth position. The single letter $w_{i:i}$ is denoted by w_i. Let h be the homomorphism defined as follows: $h([_1) = h([_2) = [_1$ and $h(]_1) = h(]_2) =]_1$. Let $w \in D_2$, $1 \leqq i < j \leqq |w|$, and let w_i be a left bracket (i.e., $w_i = [_1$ or $[_2$). (i, j) is a *matched pair* if w_i and w_j are compatible (i.e., if $w_i = [_1$ then $w_j =]_1$; if $w_i = [_2$ then $w_j =]_2$) and $h(w_{i:j})$ is balanced and irreducible.

LEMMA 6. *A string w is in D_2 if and only if the following conditions hold*: (i) $h(w) \in D_1$; (ii) *for each $1 \leqq i \leqq |w|$, if w_i is a left bracket, then there exists a j such that $i < j \leqq |w|$ and (i, j) is a matched pair.*

The proof is an induction on the length of a string. See [IBAR88] for more details.

Using Lemma 6, we have the following theorem.

THEOREM 2. *D_2 can be accepted by an alternating* 1*-reversal* 1CM.

Proof. We describe an alternating 1-reversal 1CM M accepting D_2. Given an input w, M first generates a process to verify that $h(w) \in D_1$, i.e., w has an equal number of left and right brackets, and in any prefix of w, the number of left brackets is greater than or equal to the number of right brackets. Then M moves the input head to the right and, for each symbol w_i scanned by the input head, $1 \leqq i \leqq |w|$, it verifies that if w_i is a left bracket then there exists a j, $i < j \leqq |w|$, such that (i, j) is a matched pair, i.e., w_i and w_j are compatible, $w_{i:j}$ has an equal number of left and right brackets, and in any proper prefix of $w_{i:j}$, the number of left brackets is greater than the number of right brackets.

Suppose that the input head is looking at a left bracket w_i and the counter is zero. A crucial step in the above construction is that for each j, $i \leqq j \leqq |w|$, M can check

whether the number of left brackets is greater than (=, or $<$) the number of right brackets in $w_{i:j}$. This is done as follows. M sets the counter to $|w|-i$ (by guessing and verifying). Then while moving the input head to the right, it decrements the counter by 2 for each left bracket scanned. Let c_j be the counter value when the input head leaves w_j, $i \leqq j \leqq |w|$. It is easy to see that the number of left brackets is greater than (=, or $<$) the number of right brackets in $w_{i:j}$ if and only if $|w|-j>(=, \text{or} <)\ c_j$. The latter fact can be verified easily. □

Let $G'=\langle V, \Sigma', P', S\rangle$ be an arbitrary context-free grammar (CFG). A parenthesis CFG [LYNC77] G induced by G' is $G=\langle V, \Sigma, P, S\rangle$ where $\Sigma=\Sigma' \cup \{(,)\}$ and $P=\{A \rightarrow (\alpha) | A \rightarrow \alpha \text{ is in } P'\}$. (Assume that "(" and ")" are not in Σ'.) A parenthesis CFL is a language generated by a parenthesis CFG. It was shown in [BUSS87] that parenthesis CFL's are in ATIME ($\log n$). We show that parenthesis CFL's are accepted by alternating 1-reversal 1CM's.

THEOREM 3. *Every parenthesis* CFL *can be accepted by an alternating* 1-*reversal* 1CM.

Proof (sketch). Let L be a parenthesis CFL generated by $G=\langle V, \Sigma, P, S\rangle$. We construct an alternating 1-reversal 1CM M to accept L. Given an input w, M verifies that $S \overset{*}{\Rightarrow} w$ in a top-down fashion. The process is recursive and described informally below.

Let $w_{i:j}$ be a substring of w. Suppose that $A \overset{*}{\Rightarrow} w_{i:j}$. Then $w_i=$ "(" and $w_j=$ ")," and $w_{i:j}$ is of the form $(x_0(\alpha_1)x_1 \cdots (\alpha_k)x_k)$, where the x_s's are strings over $\Sigma-\{(,)\}$, the α_s's are strings over Σ, and there exist $A_1, \cdots, A_k \in V$ such that $A_s \overset{*}{\Rightarrow} (\alpha_s)$, $1 \leqq s \leqq k$. The verification of $A \overset{*}{\Rightarrow} w_{i:j}$ can be done by guessing the A_s's and the ending position of each (α_s), and verifying that $A_s \overset{*}{\Rightarrow} (\alpha_s)$, $1 \leqq s \leqq k$, and $A \rightarrow (x_0A_1x_1 \cdots A_kx_k) \in P$. Note that the parentheses in each (α_s) form an irreducible balanced string. Thus the correctness of the guessed ending position of each (α_s) can be verified as in the proof of Theorem 2. We leave the details to the reader. □

Now we consider the linear CFL's. It is well known that linear CFL's are exactly the languages accepted by nondeterministic 1-reversal PDA's [HOPC79]. We show that linear CFL's are accepted by alternating 1-reversal 1CM's.

THEOREM 4. *Every linear* CFL *can be accepted by an alternating* 1-*reversal* 1CM.

Proof (sketch). Let $G=\langle V, \Sigma, P, A_0\rangle$ be a linear CFG. The alternating 1-reversal 1CM M accepting the language generated by G operates as follows. Let w be a string in the language. Suppose that $A_0 \Rightarrow x_1A_1y_1 \Rightarrow x_1x_2A_2y_2y_1 \Rightarrow \cdots x_1 \cdots x_mA_my_m \cdots y_1 \Rightarrow x_1 \cdots x_mzy_m \cdots y_1 = w$ is a derivation of w from A_0. Let $d_i=|x_i|$ and $e_i=|y_i|$, $1 \leqq i \leqq m$. For convenience, let $d_0=e_0=0$ and $n=|w|$. M first guesses x_1, y_1, and A_1 such that $A_0 \rightarrow x_1A_1y_1 \in P$, and verifies that $w_{d_0+1:d_1}=x_1$ and $w_{n-e_1+1:n-e_0}=y_1$. (By convention, let $w_{i:j}=\varepsilon$ if $i>j$.) Then M shifts the input head d_1 cells to the right and increments the counter by e_1. It guesses x_2, y_2, and $A_2, \cdots$ and so on. Generally, M guesses x_i, y_i, and A_i such that $A_{i-1} \rightarrow x_iA_iy_i \in P$, and verifies that $w_{d_1+\cdots+d_{i-1}+1:d_1+\cdots+d_i}=x_i$ and $w_{n-e_1-\cdots-e_i+1:n-e_1-\cdots-e_{i-1}}=y_i$. Note that, at this moment, the input head is scanning $w_{d_1+\cdots+d_{i-1}}$ and the counter value is $e_1+\cdots+e_{i-1}$. Thus, the above verifications can be done easily. M then moves the input head d_i cells to the right and increments the counter by e_i. Finally, M guesses z such that $A_m \rightarrow z \in P$ and verifies that $w_{d_1+\cdots+d_m+1:n-e_1-\cdots-e_m}=z$.

4. Alternating 1-reversal pushdown automata. Let NPDA (APDA) denote the class of languages accepted by nondeterministic (alternating) PDA's, and NPDA (k) (APDA (k)) denote the class of languages accepted by nondeterministic (alternating) k-reversal PDA's. It is well known that NPDA (1) $\subsetneq$ NPDA (3) $\subsetneq$ NPDA (5) $\subsetneq \cdots \subsetneq$ and $\bigcup_{k \geqq 1}$ NPDA (k) $\subsetneq$ NPDA. For the case of alternating finite-reversal PDA's, we

prove that a similar hierarchy does not exist. In fact, we show that APDA (1) = APDA. Since APDA $= \bigcup_{c>0}$ DTIME (c^n) [CHAN81b], [LADN84], the above result implies that APDA (1) $= \bigcup_{c>0}$ DTIME (c^n).

We will give a direct simulation of alternating PDA's by 1-reversal alternating PDA's. The definition of an alternating PDA is the same as the one in [LADN84] except that the input head movement is one-way, from left to right, and the acceptance of an input is by empty pushdown store. (It is easy to show that the two notions of acceptance are equivalent.) Formally, an alternating PDA is a 7-tuple $M = (Q, q_0, U, \Sigma, \Gamma, Z_0, \delta)$, where

Q is a finite set of states,
$q_0 \in Q$ is the start state,
$U \subseteq Q$ is the set of universal states,
Σ is the input alphabet ($\$ \notin \Sigma$),
Γ is the pushdown store alphabet,
$Z_0 \in \Gamma$ is the start symbol of the pushdown store,
δ is the transition function from $Q \times (\Sigma \cup \{\$\}) \times \Gamma$ to the finite subsets of $Q \times \{0, 1\} \times (\Gamma^2 \cup \Gamma \cup \{\varepsilon\})$, where ε represents the empty string.

Again, the input is delimited on the right by the marker \$. If the machine is in state q, scanning a on the input tape and Z on the top of the pushdown store, and if $(q', d, \gamma) \in \delta(q, a, Z)$, then the machine can enter state q', move the input head d squares to the right, and replace the top symbol Z by the string γ. We say that the machine is *pushing* if $|\gamma| > 1$, *idling* if $|\gamma| = 1$, and *popping* if $|\gamma| = 0$.

A configuration of M is a 4-tuple (q, x, i, α), where
$q \in Q$ is the current state,
$x \in \Sigma^*$ is the input,
i, where $0 \leqq i \leqq |x|$ is the distance between the input head and the right endmarker \$,
$\alpha \in \Gamma^*$ is the contents of the pushdown store, where the rightmost symbol is the top of the pushdown store.

The set of accepting configurations are those of the form (q, x, i, ε). We say that M is k reversal-bounded (or, M is an alternating k-reversal PDA) if M makes at most k reversals (i.e., alternations from pushing to popping and vice versa) on all inputs. The following theorem shows that the restriction on the number of reversals does not affect the power of alternating PDA's.

THEOREM 5. APDA (1) = APDA.

Proof. If suffices to show that APDA $\subseteq$ APDA (1), i.e., every alternating PDA can be simulated by an alternating 1-reversal PDA.

Let $M = (Q, q_0, U, \Sigma, \Gamma, Z_0, \delta)$ be an alternating PDA. To simplify the proof, we make the following (nonessential) assumption: if D belongs to the range of δ, then either (i) $D \subseteq Q \times \{0, 1\} \times \Gamma^2$, (ii) $D \subseteq Q \times \{0, 1\} \times \Gamma$, or $D \subseteq Q \times \{0, 1\} \times \{\varepsilon\}$. That is, when M makes a branch, the operations of its pushdown store are either (i) all pushings, (ii) all idlings, or (iii) all poppings. Thus the configurations of M can be divided into three groups: pushing configurations, idling configurations, and popping configurations, according to the type of the pushdown store operations in the moves associated with them.

Let x be an input and $\pi(x)$ be the computation tree of M on x. The simulating alternating 1-reversal PDA M' simulates M on x by exploring $\pi(x)$. The idea is as

follows. Denote the level (i.e., height) of the pushdown store of M at a node c by $\text{lev}(c)$. Suppose that M' is looking at some node c (of $\pi(x)$) which is a pushing configuration, and intending to explore a subtree rooted at c. Denote the subtree by T_c. Suppose $\text{lev}(c) \leqq \text{lev}(c')$ for all c' of T_c, which is a pushing configuration. If $\text{lev}(c) < \text{lev}(c')$ for all $c' \neq c$ of T_c, which is a pushing configuration, M' simulates M faithfully. Otherwise let $\text{VALLEY}_c = \{c' \mid c'$ of T_c is a pushing configuration, $\text{lev}(c') = \text{lev}(c)$, and for all c'' of T_c, if $c'' \neq c$ is an ancestor of c' and $\text{lev}(c'') = \text{lev}(c')$, then c'' must be an idling configuration$\}$. (Intuitively, VALLEY_c is the set of nodes of T_c where the pushdown store of M returns to the same level as at c for the first time.) M' universally generates $|\text{VALLEY}_c|+1$ processes. For each $c' \in \text{VALLEY}_c$, a process is assigned to the task of exploring the subtree (of T_c) rooted at c'. The additional process is for exploring the subtree obtained from T_c by pruning the subtrees rooted at the members of VALLEY_c. Note that in each of the new subtrees, the pushdown store of M at the root is of the lowest level among all the pushing nodes (configurations). In this way, $\pi(x)$ is divided into a bunch of subtrees such that M makes at most one reversal in each of them, and it is then explored directly. The difficulty is in how to make sure that the explorations of those subtrees are combined correctly.

Now we give the details of M'. The set of states of M' is a superset of the set of states of M. Assume that $Q \cap \Gamma = \emptyset$, and $\#$, $\downarrow$, Z_0' are new symbols not in Γ. Then the pushdown store alphabet of $M' = \Gamma \cup Q \cup \{\#, \downarrow, Z_0'\}$, where Z_0' is the start symbol, $\#$ is the "counting" symbol, and $\downarrow$ is the "cancellation" symbol. The uses of $\#$ and $\downarrow$ will be discussed later. The operation of M' is described by the following procedure. In the procedure, we use the variables STATE, HEAD, and TOP to denote the current state, the distance between the input head and the right endmarker \$, and the symbol on the top of the pushdown store, respectively. If $C = (q, x, i, \alpha)$ is a configuration of M' with $q \in Q$, then C_M denotes (q, x, i, α_M), the corresponding configuration of M, where α_M is obtained by erasing the symbols that do not belong to Γ.

```
                procedure SIMULATE (x);
                  {Simulate M on x.}
                  Push Z0' onto the pushdown store;
                  STATE := q0;
UPHILL:           {Let C = (q, x, i, α) be the current configuration and
                  C_M = (q, x, i, α_M) be the corresponding
                  configuration of M. Explore the subtree of π(x) rooted
                  at C_M until a popping configuration is encountered.
                  Let Z = TOP be the current top symbol.}
                case of C_M :
                  popping:  goto DOWNHILL;
                   idling:  Simulate M for one step;
                            goto UPHILL;
                  pushing: loop
                           do existentially
                              (i) Simulate M for one step;
                                  goto UPHILL;
                             (ii) Replace Z by some #^j q'Z,
                                                          where q' ∈ Q, 0 ≤ j ≤ i,
                                  and (q', x, j, α_M) is
                                       a pushing configuration of M;
                                  do universally
```

```
                    (a) continue;
                    (b) Replace Z by ↓Z;
                        HEAD := j' for some j' ≦ i;
                        do universally
                          (b1) Verify HEAD = j;
                          (b2) STATE := q';
                               goto UPHILL;
                        end
                end
           end
         endloop
       endcase;

DOWNHILL:  {Let C and C_M be as in UPHILL. Explore the subtrees of
             π(x) rooted at C_M until a pushing or accepting
             configuration is reached. Let q = STATE be the current state.}
       case of C_M
         popping: Simulate M for one step;
                  if TOP ∉ Γ then pop until some Z ∈ Γ
                  is on the top of the pushdown store;
                  goto DOWNHILL;
          idling: Simulate M for one step;
                  goto DOWNHILL;
         pushing: Pop;
                  loop
                    if TOP ∉ Q then reject;
                    do existentially
                       (i) Pop until TOP ≠ # again;
                      (ii) Verify that TOP=q and HEAD = number
                           of #'s (in the pushdown store) following
                           the top symbol;
                           Pop until the pushdown store is empty;
                    end
                  endloop
       endcase
       end. {The end of SIMULATE.}
```

It should be noticed that, in procedure SIMULATE, "simulate M for one step" means that M' universally (existentially) branches if M universally (existentially) branches. It is not hard to show that the procedure can be implemented on an alternating 1-reversal PDA.

Procedure SIMULATE consists of two blocks: UPHILL and DOWNHILL. For a given input x, UPHILL is used to explore the subtrees of $\pi(x)$ consisting of pushing and idling nodes (configurations). During the exploration, every time M' is looking at a pushing node (call it c), it guesses whether there are some pushing nodes where the pushdown store of M returns to the same level as at c for the first time. If M' guesses that there are no such nodes, it faithfully simulates M for one step. Otherwise, it guesses those nodes and does the following for each of them nondeterministically, one at a time, and then simulates M for one step. Let c' be a node that M' has guessed. First it guesses and records the configuration of M at c'. Since the pushdown store of M at c' is the same as at c, only the input head position (in the form of $\#^i$) and state

(in the form of q) need to be pushed onto the pushdown store. Then it universally generates two processes. The first process is to continue the current work at c. When this process reaches c', the input head position and state of M at c' must be at the top of the pushdown store and it accepts. The second process is to initiate the exploration starting at c'. Since a configuration may be repeated many times, there is a danger that the second process reaches some node where the configuration of M is the same as the one at c', and accepts because the input head position and state of M at c' are stored at the top of the pushdown store. This may lead to the acceptance of some inputs which are not accepted by M. To resolve this problem, before the exploration starting at c' is initiated, the marker $\downarrow$ is attached to the segment (i.e., $\#^i q$) representing the input head position and state of M at c', to indicate that the segment has been "canceled." Note that since M' can make at most one reversal, it cannot simply erase $\#^i q$.

DOWNHILL is used for exploring the subtrees of $\pi(x)$ consisting of popping and idling nodes. When M' is simulating a popping move of M, the symbols that do not belong to Γ are considered as "garbage" and erased whenever they appear on the top of the pushdown store. When it encounters a pushing node, M' verifies that the input head position and state of M at that node are consistent with the ones (which have not been canceled, i.e., no $\downarrow$ is attached) stored at the top of the pushdown store. If there are several input head position and state pairs stored at the top of the pushdown store consecutively (i.e., not separated by some $Z \in \Gamma$), it nondeterministically chooses one and checks the consistency.

The correctness of SIMULATE can be verified by using induction on the number of reversals that M makes in a computation tree. □

It is known that APDA = two-way APDA (= the class of languages accepted by alternating two-way PDA's) = ASPACE $(n) = \bigcup_{c>0}$ DTIME (c^n) [CHAN81b], [LADN84]. From Theorem 5, we have Corollary 2.

COROLLARY 2. APDA (1) = *two-way* APDA = ASPACE $(n) = \bigcup_{c>0}$ DTIME (c^n).

REFERENCES

[BAKE74] B. BAKER AND R. BOOK, *Reversal-bounded multipushdown machines*, J. Comput. System Sci., 8 (1974), pp. 315–332.

[BUSS87] S. BUSS, *Boolean formula-value problem is in* ALOGTIME, Proc. 19th Annual ACM Symposium on Theory of Computing, New York, 1987, pp. 123–131.

[CHAN81a] T. CHAN, *Reversal complexity of counter machines*, Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, 1981, pp. 146–157.

[CHAN81b] A. CHANDRA, D. KOZEN, AND L. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[CHEN87] J. CHEN AND C. YAP, *Reversal complexity*, Proc. 2nd Annual Conference on Structure in Complexity Theory, Cornell University, Ithaca, NY, 1987, pp. 14–19.

[FISH68] P. FISHER, *The reduction of tape reversal for off-line one-tape Turing machines*, J. Comput. System Sci., 2 (1968), pp. 136–147.

[GURA81] E. GURARI AND O. IBARRA, *The complexity of decision problems for finite-turn multicounter machines*, J. Comput. System Sci., 22 (1981), pp. 220–229.

[GREI73] S. GREIBACH, *The hardest context-free language*, SIAM J. Comput., 2 (1973), pp. 304–310.

[HART68] J. HARTMANIS, *Tape-reversal bounded Turing machine computations*, J. Comput. System Sci., 2 (1968), pp. 117–135.

[HONG80] J. HONG, *On similarity and duality of computation*, Proc. 21st IEEE Symposium on Foundations of Computer Science, Syracuse, NY, 1980, pp. 348–359.

[HOPC79] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[HROM85] J. HROMKOVIC, *Alternating multicounter machines with constant number of reversals*, Inform. Process. Lett., 21 (1985), pp. 7–9.

[IBAR78] O. IBARRA, *Reversal-bounded multicounter machines and their decision problems*, J. Assoc. Comput. Mach., 25 (1978), pp. 116–133.

[IBAR88] O. IBARRA, T. JIANG, AND B. RAVIKUMAR, *Some subclasses of context-free languages in* NC^1, Inform. Process. Lett., 29 (1988), pp. 111–117.

[KAME70] T. KAMEDA AND R. VOLLMAR, *Note on tape reversal complexity of languages*, Inform. and Control, 17 (1970), pp. 203–215.

[LADN84] R. LADNER, R. LIPTON, AND L. STOCKMEYER, *Alternating pushdown automata*, SIAM J. Comput., 13 (1984), pp. 135–155.

[LYNC77] N. LYNCH, *Log space recognition and translation of parenthesis languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 583–590.

[PIPP79] N. PIPPENGER, *On simultaneous resource bounds*, Proc. 20th IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1979, pp. 307–311.

[RYTT85] W. RYTTER AND M. CHROBAK, *A characterization of reversal-bounded multipushdown machine languages*, Theoret. Comput. Sci., 36 (1985), pp. 341–344.

[YAMA87] H. YAMAMOTO AND S. NOGUCHI, *Comparison of the power between reversal-bounded* ATM*s* *and reversal-bounded* NTM*s*, Inform. and Comput., 75 (1987), pp. 144–161.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 291-314, April 1991
© 1991 Society for Industrial and Applied Mathematics
008

INTERPOLATION AND APPROXIMATION OF SPARSE MULTIVARIATE POLYNOMIALS OVER $GF(2)$*

RON M. ROTH† AND GYORA M. BENEDEK‡

Abstract. A function $f:\{0, 1\}^n \rightarrow \{0, 1\}$ is called t-sparse if the n-variable polynomial representation of f over $GF(2)$ contains at most t monomials. Such functions are uniquely determined by their values at the so-called critical set of all binary n-tuples of Hamming weight $\geqq n - \lfloor \log_2 t \rfloor - 1$. An algorithm is presented for interpolating any t-sparse function f, given the values of f at the critical set. The time complexity of the proposed algorithm is proportional to n, t, and the size of the critical set. Then, the more general problem of approximating t-sparse functions is considered, in which case the approximating function may differ from f at a fraction ε of the space $\{0, 1\}^n$. It is shown that $O((t/\varepsilon) \cdot n)$ evaluation points are sufficient for the (deterministic) ε-approximation of any t-sparse function, and that an order of $(t/\varepsilon)^{\alpha(t,\varepsilon)} \cdot \log n$ points are necessary for this purpose, where $\alpha(t, \varepsilon) \geqq 0.694$ for a large range of t and ε. Similar bounds hold for the t-term DNF case as well. Finally, a probabilistic polynomial-time algorithm is presented for the ε-approximation of any t-sparse function.

Key words. interpolation of sparse polynomials, approximation of sparse polynomials, learning of Boolean functions, Reed-Muller codes

AMS(MOS) subject classifications. 41A05, 41A10, 68Q20, 68R99, 94B35

1. Introduction. Consider a Boolean function $f:\{0, 1\}^n \rightarrow \{0, 1\}$ which maps a vector $[x_{n-1}x_{n-2} \ldots x_0]$ into $f(x_{n-1}, x_{n-2}, \ldots, x_0)$. One common way to classify such functions is by the minimal number t for which there exists a *t-term disjunctive normal form* (in short, t-term DNF) expression equivalent to f; that is, an expression consisting of an inclusive OR of up to t products (AND) of variables, each variable possibly complemented (NOT). Another way of classifying Boolean functions is by the number of nonzero monomials in the (unique) n-variable polynomial representation of f over $GF(2)$,

$$f(x_{n-1}, x_{n-2}, \ldots, x_0) = \sum_{i=0}^{2^n-1} f_i \cdot x_{n-1}^{i_{n-1}} x_{n-2}^{i_{n-2}} \cdots x_0^{i_0}, \tag{1}$$

where $\mathbf{i} \triangleq [i_{n-1}i_{n-2} \ldots i_0]$ is the n-bit binary representation of i, and summation is carried out over $GF(2)$ (XOR). An n-variable Boolean function f is *t-sparse* if the number of nonzero monomials in the polynomial representation (1) of f is at most t.

In this work we first address the problem of *interpolating* t-sparse functions, that is: Given n, t, and the values of a t-sparse function f at a subset P of $\{0, 1\}^n$, can f be determined uniquely? If so, is there an efficient algorithm (i.e., in time complexity polynomial in n, t, and $|P|$), by which f can be retrieved?

These questions arise in several applications, e.g., in the study of function learnability and inductive inference [1], [2], [13]-[15]. In this model, a "student" tries to "learn" an underlying function f, given the values of f at some set of points $P \subseteq \{0, 1\}^n$. Knowing the value of n (and, sometimes, t), the question is whether the student can retrieve f efficiently out of its values at P.

In § 2 we show that, for the unique interpolation of f, the set P must contain a "critical set" consisting of all binary n-tuples of Hamming weight $\geqq n - \lfloor \log_2 t \rfloor - 1$.

* Received by the editors July 5, 1989; accepted for publication (in revised form) May 30, 1990.

† IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120, and Computer Science Department, Technion-Israel Institute of Technology, Haifa 32000, Israel.

‡ Rosh Intelligent Systems Ltd., P.O. Box 03552, Mevasseret Zion 90805, Israel. This work was done while the author was with the Computer Science Department, Technion-Israel Institute of Technology, Haifa 32000, Israel.

This result applies to the *nonadaptive setting*, where the points of evaluation do not depend on values of f at previously-queried points. It turns out that adaptive schemes do not yield any significant reduction in the number of necessary queries. The existence of such a critical set has been proved (independently) also by Clausen et al. in [4]. Our result is somewhat stronger, showing that finding the *parity* of the truth table of f requires at least as many evaluation points as required for finding the truth table itself.

In § 3 we present a deterministic nonadaptive algorithm which retrieves the underlying function f out of its values at this critical set in $O(t \cdot n \cdot \sum_{i=0}^{1+\lfloor \log_2 t \rfloor} \binom{n}{i})$ bit operations. Establishing the correspondence between the interpolation problem and the decoding of certain error-correcting codes, our interpolation algorithm may also serve as a [syndrome-based] decoding algorithm for Reed–Muller codes [9, Chap. 13]. We conclude our interpolation discussion by showing that, for fixed $t > 1$, deciding whether there exists a t-sparse function passing via a given (arbitrary) set of evaluation points is NP-complete (§ 4).

Interpolation algorithms have been presented also by Ben-Or and Tiwari [3], Clausen et al. [4], and Grigoriev, Karpinski, and Singer [6]; however, in their model, f is evaluated at n-tuples over an *extension field* $GF(2^m)$ (in which case t evaluation points can be shown to be sufficient), whereas in our case the evaluation points are confined to n-tuples over the ground field $GF(2)$. This extension field model has been motivated, in part, by the fact that the size of the critical set is nonpolynomial in n and t.

Another way of overcoming the nonpolynomial nature of Boolean interpolation is by considering the more general problem of *approximating* Boolean functions. In this scheme, we may end up with a function $\hat{f}$ whose truth table differs from that of f at less than $\varepsilon \cdot 2^n$ entries for some (prespecified) $0 < \varepsilon \leqq 1$. This scheme is widely used in the context of function learnability, with the functions usually being represented as DNF expressions.

Much work has been done on the (still unresolved) problem of finding an efficient algorithm for the t-term DNF approximation [8], [11], [13]-[15]. The last two sections in this paper are devoted to the t-sparse polynomial approximation problem. In § 6 we present an approximation algorithm which, given n, t, ε, and a (small) probability p of failure, finds an ε-approximation for any t-sparse function with probability $\geqq 1 - p$, requiring $O((t^2 n^2/\varepsilon) \cdot \log(tn/p))$ bit operations. We believe that this result may shed light on the t-term DNF approximation problem as well, and it exhibits one of the advantages of the polynomial representation in studying the learnability of Boolean functions.

Preceding the presentation of the above algorithm, we obtain in § 5 lower and upper bounds on the number of evaluation points required for the approximation of t-sparse functions. We show that $O((t/\varepsilon) \cdot n)$ points are sufficient for the (deterministic) ε-approximation of any t-sparse function, and that an order of $(t/\varepsilon)^{\alpha(t,\varepsilon)} \cdot \log n$ points are necessary for this purpose, where $\alpha(t, \varepsilon) \geqq 0.694$ for a large range of t and ε (Thms. 5.1, 5.3). Similar bounds are derived for the t-term DNF case as well.

2. Background and basic results. Given a function f over $\{0,1\}^n$, let $\mathbf{f} \triangleq [f_0 f_1 \ldots f_{2^n-1}]'$ denote the column vector of coefficients of f as defined by (1) and let

$$\mathbf{F} = \begin{bmatrix} f(0,\ldots,0,0,0) \\ f(0,\ldots,0,0,1) \\ f(0,\ldots,0,1,0) \\ \vdots \\ f(1,\ldots,1,1,1) \end{bmatrix}$$

denote the truth table of f. Let A be the 2×2 matrix given by

$$A \triangleq \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},$$

and define the $2^n \times 2^n$ matrix A_n as follows: $A_0 \triangleq [1]$ and, for $n \geqq 1$, $A_n \triangleq A \otimes A_{n-1}$, where $\otimes$ stands for the direct (or Kronecker) product of matrices. For instance,

$$A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Writing f as a polynomial in x_{n-1},

$$(2) \qquad f(x_{n-1}, x_{n-2}, \ldots, x_0) = f_0(x_{n-2}, x_{n-3}, \ldots, x_0) + x_{n-1} \cdot f_1(x_{n-2}, x_{n-3}, \ldots, x_0),$$

it is easy to show by induction on n that for every function f over $\{0, 1\}^n$, $\mathbf{F} = A_n\mathbf{f}$ [9, Chap. 13, § 2]. Also, since $A_n = A_n^{-1}$, we have $\mathbf{f} = A_n\mathbf{F}$ and, in particular, the parity of the number of 1's in $\mathbf{F}$ is equal to the coefficient of $x_{n-1}x_{n-2} \cdots x_0$ in the polynomial representation of f.

Remark 2.1. It is sometimes convenient to use the following equivalent definition of A_n. Given two binary n-vectors $\mathbf{i} = [i_{n-1}i_{n-2} \ldots i_0]$ and $\mathbf{j} = [j_{n-1}j_{n-2} \ldots j_0]$, we say that $\mathbf{j}$ is *dominated* by $\mathbf{i}$, denoted $\mathbf{j} \sqsubseteq \mathbf{i}$, if $j_k \leqq i_k$ for all $0 \leqq k \leqq n-1$. It can be readily verified that $A_n[i, j] = 1$ if and only if $\mathbf{j} \sqsubseteq \mathbf{i}$, with $\mathbf{i}$ and $\mathbf{j}$ standing for the binary representations of i and j, $0 \leqq i, j \leqq 2^n - 1$ [9, Chap. 13, § 2].

The t-sparse interpolation problem can now be formulated in the following coding theory terms. Assume that the values of f are given at some l points in $\{0, 1\}^n$. These values can be written as a binary l-tuple $\mathbf{s} \triangleq H\mathbf{f}$, known as the *syndrome* of $\mathbf{f}$, where H is an $l \times 2^n$ submatrix of A_n. The interpolation process can now be viewed as the *decoding* of the vector $\mathbf{f}$ given the vector $\mathbf{s}$. In order to achieve unique interpolation, every $2t$ columns of H must be linearly independent, or else there would be two distinct t-sparse functions $\mathbf{f}_1$ and $\mathbf{f}_2$ such that $H\mathbf{f}_1 = H\mathbf{f}_2$. On the other hand, if every $2t$ columns of H are linearly independent, then $\mathbf{s}$ determines $\mathbf{f}$ uniquely, provided the latter is t-sparse. Therefore, H must be a *parity-check matrix* of a binary linear code of length 2^n, dimension $\geqq 2^n - l$, and minimum distance $\geqq 2t+1$.

The above discussion leads us to the well-known relation between the interpolation problem and Reed–Muller codes [9, Chap. 13], which we briefly summarize below. For every $\mathbf{u} \in \{0, 1\}^n$, denote by $w(\mathbf{u})$ the Hamming weight of $\mathbf{u}$. Let $S(n, r)$ be the set of all vectors $\mathbf{u} \in \{0, 1\}^n$ with $w(\mathbf{u}) \geqq n-r$ and let $V(n, r) \triangleq |S(n, r)| = \sum_{i=0}^{r} \binom{n}{i}$ (when $r > n$ or $r < 0$ we define $\binom{n}{r} \triangleq 0$). From the properties of the Pascal triangle it is easy to verify the identity $V(n, r) = V(n-1, r) + V(n-1, r-1)$.

Now, let $H_{n,r}$ be the binary $V(n, r) \times 2^n$ matrix consisting of the rows of A_n whose indices i are of binary representation $\mathbf{i} \in S(n, r)$, with the order of these rows maintained

as in A_n. It is easy to verify that

$$H_{n,r} = \begin{bmatrix} H_{n-1,r-1} & 0 \\ H_{n-1,r} & H_{n-1,r} \end{bmatrix}. \tag{3}$$

The matrix $H_{n,r}$ is known as the parity-check matrix of the binary $(n-r-1)$st order Reed–Muller code of length 2^n, the minimum distance of which is 2^{r+1}. The next lemma is a direct corollary of the known properties of Reed–Muller codes.

LEMMA 2.1. *For $1 \leqq t \leqq 2^n$, the $V(n, 1+\lfloor \log_2 t \rfloor)$ values of a t-sparse function $f:\{0,1\}^n \to \{0,1\}$ at $S(n, 1+\lfloor \log_2 t \rfloor)$ are sufficient in order to determine uniquely any such function f.*

Proof. This follows from the fact that every $2t$ columns in $H_{n,1+\lfloor \log_2 t \rfloor}$ are linearly independent. □

The following lemma is the converse of Lemma 2.1. Moreover, we show that finding the coefficient of $x_{n-1}x_{n-2}\cdots x_0$ in f requires as many evaluation points as required for finding f itself.

LEMMA 2.2. *In order to find the parity of the truth table of any t-sparse function $f:\{0,1\}^n \to \{0,1\}$, $1 \leqq t \leqq 2^n$, the values of f should be specified at all points of $S(n, 1+\lfloor \log_2 t \rfloor)$.*

Proof. First, $[1\ 1 \ldots 1]$ is the only evaluation point distinguishing the zero function from $x_{n-1}x_{n-2}\cdots x_0$. Now, let $0 < r \leqq 1+\lfloor \log_2 t \rfloor$ and assume that $\mathbf{z} \triangleq [\underbrace{00\ldots0}_{r}\,11\ldots1]$ is not one of the evaluation points. Denote by $\bar{x}_j$ the complement of the variable x_j, and define the functions f and g by

$$f(x_{n-1}, x_{n-2}, \ldots, x_0) \triangleq \bar{x}_{n-2}\bar{x}_{n-3}\cdots\bar{x}_{n-r}\cdot x_{n-r-1}x_{n-r-2}\cdots x_0 \tag{4}$$

and

$$g(x_{n-1}, x_{n-2}, \ldots, x_0) \triangleq x_{n-1}\cdot\bar{x}_{n-2}\bar{x}_{n-3}\cdots\bar{x}_{n-r}\cdot x_{n-r-1}x_{n-r-2}\cdots x_0. \tag{5}$$

It is easy to see that the (nonzero) sum $f+g$ $(=\bar{x}_{n-1}\cdot f)$ vanishes at all points of $\{0,1\}^n$ except $\mathbf{z}$. Substituting $1+x_j$ for $\bar{x}_j$ in the right-hand sides of (4) and (5), and expanding the expressions thus obtained, we conclude that f and g are both t-sparse functions taking the same value at every evaluation point. On the other hand we have $w(\mathbf{F})=2$, whereas $w(\mathbf{G})=1$. □

We can therefore summarize with the following theorem.

THEOREM 2.1. *For $1 \leqq t \leqq 2^n$, the $V(n, 1+\lfloor \log_2 t \rfloor)$ values of a t-sparse function $f:\{0,1\}^n \to \{0,1\}$ at $S(n, 1+\lfloor \log_2 t \rfloor)$ are necessary and sufficient in order to determine uniquely any such function f.*

We now turn to the adaptive case, where the points of evaluation may depend on values of the underlying function at previously queried points. In such a scheme, any interpolation procedure can be described in a form of a tree: Each vertex corresponds to an interpolation query, whose result determines which one of the successive subtrees we should pick next. Each leaf in the tree is associated with at most one n-variable function, and every t-sparse function must be associated with at least one leaf.

Let v_0 be a leaf corresponding to the zero function, let l_0 be its distance from the root, and let P_0 denote the l_0 interpolation points queried from the root up to v_0. For unique interpolation, none of the nonzero t-sparse functions should vanish at P_0. Following similar arguments, as given in the proof of Lemma 2.2, we obtain the lower bound $l_0 \geqq V(n, \lfloor \log_2 t \rfloor)$. This leaves quite a marginal benefit, if any, in using adaptive interpolation algorithms, compared with the nonadaptive case.

3. Interpolation algorithm for *t*-sparse functions. There exists a well-known decoding algorithm for Reed–Muller codes, based on majority logic circuits [9, Chap. 13,

§§ 6, 7]. However, this algorithm is not suitable for our purposes, as its time complexity is proportional to 2^n. Instead, we describe a recursive procedure for solving deterministically the n-variable t-sparse interpolation problem: given the values of a t-sparse function $f:\{0,1\}^n \to \{0,1\}$ at the critical set specified in Theorem 2.1, the algorithm retrieves f in $O(t \cdot n \cdot V(n, 1+\lfloor \log_2 t \rfloor))$ bit operations. A similar algorithm has been discovered recently by Hellerstein and Warmuth [7] also.

The procedure, named INTERPOL, is presented in Fig. 1. Given an underlying n-variable t-sparse function f, the input to INTERPOL consists of the number of variables n, an integer r such that $t \leqq 2^r - 1$, and the vector $\mathbf{s} = H_{n,r}\mathbf{f}$ of values of f at $S(n, r)$. The output of INTERPOL is the support F of the coefficient vector $\mathbf{f}$, i.e., the set of indices of the nonzero entries of $\mathbf{f}$.

The first steps of INTERPOL check whether either r or n is zero, in which case $\mathbf{s}$ is a scalar and, therefore, f can be determined in a straightforward manner. Note that when $r = 0$ and $\mathbf{s} \neq \mathbf{0}$ there is no solution for f, causing the procedure to raise the "*failure*" flag.

In case both n and r are nonzero, we enter the recursion stage. Let $\mathbf{f}_0$ and $\mathbf{f}_1$ denote the first and second halves of the coefficient vector $\mathbf{f}$, each $\mathbf{f}_j$ being a vector of

```
procedure INTERPOL(n, r, s) output: F;
/*
    Interpolation algorithm for n-variable t-sparse functions, t ≦ 2^r − 1.
    s = H_{n,r} f, where f is the t-sparse underlying (interpolated) function.
    F is the support of f, i.e., the set of indices of the nonzero entries of f.
    The procedure returns an error code "failure" in case there is no t-sparse function f satisfying s = H_{n,r} f.
*/
begin
    if r = 0 then
        if s = [0] then F ← ∅ else return "failure"
        /*  An empty set (F = ∅) corresponds to f ≡ 0.  */
    else if n = 0 then
        if s = [1] then F ← {0} else F ← ∅
        /*  F = {0} corresponds to f ≡ 1.  */
    else begin
        s_0 ← the V(n − 1, r − 1)-prefix of s;
        s_+ ← the V(n − 1, r)-suffix of s;
        F_+ ← INTERPOL(n − 1, r, s_+);
        if ("failure" while finding F_+) then return "failure"
        else begin
          F_0 ← INTERPOL(n − 1, r − 1, s_0);
          F_1 ← F_0 ⊕ F_+   /*  ≜ (F_0 ∪ F_+) − (F_0 ∩ F_+)  */;
          if |F_0| + |F_1| ≧ 2^r or ("failure" while finding F_0) then
          begin
            s_2 ← [the entries of s of indices i ≧ 2^{n−1}, i ∈ S(n, r − 1)];
            s_1 ← s_0 + s_2;
            F_1 ← INTERPOL(n − 1, r − 1, s_1);
            F_0 ← F_1 ⊕ F_+;
            if |F_0| + |F_1| ≧ 2^r or ("failure" while finding F_1) then
              return "failure"
          end;
          F ← {i | i ∈ F_0 or i − 2^{n−1} ∈ F_1}
        end
    end
end;
```

FIG. 1. *Procedure* INTERPOL.

length 2^{n-1}. The basic idea is to find $\mathbf{f}$ by computing the two vectors $\mathbf{f}_0$ and $\mathbf{f}_1$ using recursive calls to INTERPOL. In order to find these vectors, we first issue the call INTERPOL($n-1, r, \mathbf{s}_+$), where $\mathbf{s}_+$ is a vector consisting of the last $V(n-1, r)$ entries of $\mathbf{s}$. This corresponds to interpolating the $(n-1)$-variable polynomial f_+, associated with the vector $\mathbf{f}_+ \triangleq \mathbf{f}_0 + \mathbf{f}_1$ (and represented by its support F_+). Interpolation failure at this stage indicates that t is greater than $2^r - 1$.

We now perform a second call to INTERPOL with the parameters INTERPOL($n-1$, $r-1, \mathbf{s}_0$), where $\mathbf{s}_0$ is a vector consisting of the first $V(n-1, r-1)$ entries of $\mathbf{s}$. If no failure has occurred at this call, the computed set F_0 is the support of $\mathbf{f}_0$, allowing us to compute the support F_1 of $\mathbf{f}_1 = \mathbf{f}_0 + \mathbf{f}_+$ (or, in set notation, $F_1 = F_0 \oplus F_1$). The sum of the sizes of F_0 and F_1 now determines whether a third call to INTERPOL should be made. Such a call, when required, recalculates the sets F_0 and F_1. To do this, we need to compute an intermediate vector $\mathbf{s}_2$ which consists of the entries of $\mathbf{s}$ of indices $i \geqq 2^{n-1}$, $\mathbf{i} \in S(n, r-1)$ (note that $\mathbf{s}_2$ is of the same length $(V(n-1, r-1))$ as the previously obtained vector $\mathbf{s}_0$). The set F_1 is recomputed by the call INTERPOL($n-1, r-1, \mathbf{s}_1$), where $\mathbf{s}_1 = \mathbf{s}_0 + \mathbf{s}_2$, and then F_0 is updated accordingly. At this point we must have $w(\mathbf{f}) = |F_0| + |F_1| \leqq 2^r - 1$, unless t was not in the right range in the first place. Finally, the support F of $\mathbf{f}$ is obtained as the union of F_0 and a 2^{n-1}-offset of F_1.

To summarize, given n and t, the interpolation of t-sparse functions f over $\{0, 1\}^n$ is carried out first by querying the values $\mathbf{s} = H_{n,1+\lfloor \log_2 t \rfloor}$ and then calling INTERPOL using the parameters INTERPOL(n, $1 + \lfloor \log_2 t \rfloor$, $\mathbf{s}$).

LEMMA 3.1. *Let f be a t-sparse function over $\{0, 1\}^n$ and let r be an integer such that $t \leqq 2^r - 1$. Given the values $\mathbf{s} = H_{n,r}\mathbf{f}$ of f at $S(n, r)$, the output of* INTERPOL(n, r, $\mathbf{s}$) *equals the set of indices of the nonzero coefficients of f.*

Proof. Consider first the case when $r = 0$. Here t must be zero and, therefore, both $\mathbf{f}$ and $\mathbf{s}$ must be zero. Hence, if $\mathbf{s} \neq \mathbf{0}$, our assumption on the range of t is readily not satisfied, in which case INTERPOL returns "*failure.*"

Assuming from now on that $r > 0$, we continue the proof by induction on n. When $n = 0$, we have either $f \equiv 0$ or $f \equiv 1$, according to the value of the scalar $\mathbf{s}$ (note that r might be greater than n).

Now suppose that both n and r are nonzero. Recalling the definitions of $\mathbf{f}_0$, $\mathbf{f}_1$, and $\mathbf{f}_+$, by (3) we have $\mathbf{s}_0 = H_{n-1,r-1}\mathbf{f}_0$ and $\mathbf{s}_+ = H_{n-1,r}\mathbf{f}_+ = H_{n-1,r}(\mathbf{f}_0 + \mathbf{f}_1)$. Note that our assumption on t implies $w(\mathbf{f}_+) \leqq t \leqq 2^r - 1$ and, therefore, by the induction hypothesis, the execution of INTERPOL($n-1$, r, $\mathbf{s}_+$) will end up with the support F_+ of $\mathbf{f}_+$.

Second, we find either $\mathbf{f}_0$ or $\mathbf{f}_1$, and then solve for the other half. Note that at least one of these vectors must have weight $\leqq t/2 \leqq 2^{r-1} - 1$. Suppose first that $w(\mathbf{f}_0) \leqq w(\mathbf{f}_1)$. Noting that $\mathbf{s}_0 = H_{n-1,r-1}\mathbf{f}_0$, the induction hypothesis implies that the execution of INTERPOL($n-1$, $r-1$, $\mathbf{s}_0$) will result in the support F_0 of $\mathbf{f}_0$, allowing us to calculate the support F_1 of $\mathbf{f}_1$. Now, if, indeed, $w(\mathbf{f}_0) \leqq 2^{r-1} - 1$, all the above executions of INTERPOL must end successfully (i.e., without the "*failure*" flag raised) and, therefore, we must have

$$|F_0| + |F_1| = w(\mathbf{f}_0) + w(\mathbf{f}_1) \leqq 2^r - 1. \tag{6}$$

Furthermore, since there exists at most one solution $\mathbf{f}$ of weight $\leqq 2^r - 1$ to $\mathbf{s} = H_{n,r}\mathbf{f}$, the existence of a solution $\mathbf{f}_0$ for $\mathbf{s}_0 = H_{n,r}\mathbf{f}_0$, with a vector $\mathbf{f}_1 = \mathbf{f}_+ + \mathbf{f}_0$ satisfying (6), is a *sufficient* criterion for a successful interpolation of $\mathbf{f}$.

Now, suppose that (6) does not hold, or that the execution of INTERPOL($n-1$, $r-1, \mathbf{s}_0$) returns "*failure*" at one of its recursion levels. This implies that $w(\mathbf{f}_1) \leqq 2^{r-1} - 1$ and, therefore, $\mathbf{f}_1$ should be recovered successfully by INTERPOL. The corresponding vector $\mathbf{s}_1 \triangleq H_{n-1,r-1}\mathbf{f}_1$ can be found by observing that $H_{n-1,r-1}$ is a submatrix of $H_{n-1,r}$;

hence, $\mathbf{s}_1$ can be written as the sum of $\mathbf{s}_0$ and the vector $\mathbf{s}_2$ consisting of the entries of $\mathbf{s}$ of indices $i \geqq 2^{n-1}$, $\mathbf{i} \in S(n, r-1)$. Now, failure to satisfy (6) this time implies that $w(\mathbf{f}) \geqq 2^r$, in which case INTERPOL returns "*failure.*" □

THEOREM 3.1. *The interpolation of n-variable t-sparse functions can be carried out in* $O(t \cdot n \cdot V(n, 1+\lfloor \log_2 t \rfloor))$ *bit operations.*

Proof. Let $\tau(n, r)$ denote the number of bit operations required while executing INTERPOL$(n, r, \mathbf{s})$. We assume that set operations are bounded by the size of the sets times n, and that $\mathbf{s}$ is sorted according to ascending order of indices i, $\mathbf{i} \in S(n, r)$. Note that given such an index i, its successor can be evaluated by the rule $i \leftarrow i + \lceil 2^{n-r-w(i+1)} \rceil$ (where $\lceil \cdot \rceil$ stands for the ceiling function), and it takes $O(n \cdot V(n, r))$ bit operations to generate all such indices i. This rule can also be used to extract $\mathbf{s}_2$ out of $\mathbf{s}$ in $O(n \cdot V(n-1, r-1))$ bit operations. We thus have

$$\tau(n, r) = \tau(n-1, r) + 2 \cdot \tau(n-1, r-1) + O(n \cdot V(n-1, r-1)) + O(n \cdot 2^{\min(n,r)}),$$

with the initial values $\tau(0, r) = O(1)$ and $\tau(n, 0) = O(n)$. It follows by induction on n that there exists a constant β such that

$$\tau(n, r) \leqq \beta \cdot (2^{r+1} - 1) \cdot (n+1) \cdot V(n, r).$$

Hence, we conclude that the execution of INTERPOL$(n, 1+\lfloor \log_2 t \rfloor, \mathbf{s})$ involves $O(t \cdot n \cdot V(n, 1+\lfloor \log_2 t \rfloor))$ bit operations. □

4. The Interpolation Decision Problem is intractable. The time complexity of the procedure presented in § 3 is polynomial in n when t is *fixed.* This observation can be put in contrast with the next theorem which establishes the intractability of the *t-Interpolation Decision Problem* (in short, t-ID), defined as follows: Given a fixed integer t, an instance of the problem consists of an integer n and a subset $R \triangleq \{(\mathbf{v}_i; s_i)\}_{i=1}^m$ of $\{0, 1\}^n \times \{0, 1\}$. The problem is to decide whether there exists an n-variable t-sparse function f such that $f(\mathbf{v}_i) = s_i$ for all $1 \leqq i \leqq m$.

THEOREM 4.1. *For any fixed* $t > 1$, *the t-Interpolation Decision Problem is* NP-*complete.*

The proof of Theorem 4.1 is carried out, in part, by a reduction from the so-called *Hypergraph t-Colorability Problem* (in short, Hyper-t-Col) to the t-ID Problem. For fixed t, an instance of the Hyper-t-Col Problem consists of a finite set Q and a collection $\mathscr{C} = \{Q_1, Q_2, \ldots, Q_m\}$ of subsets (or *constraints*) $Q_i \subseteq Q$. The problem is to decide whether there exists a function (or *coloring*) $\chi: Q \to \{1, 2, \ldots, t\}$, such that each Q_i contains two elements y and z for which $\chi(y) \neq \chi(z)$. A similar reduction is used in [11] to show the intractability of the problem of deciding whether there exists a t-term DNF passing via a given set of points.

LEMMA 4.1 [5, p. 221], [11]. *For every fixed* $t \geqq 2$, *the Hypergraph t-Colorability Problem is* NP-*complete.*

Lemma 4.1 holds even when each Q_i in $\mathscr{C}$ is of size $\leqq 3$. Without loss of generality we can also assume that every Q_i is of size $\geqq 2$ and that the Q_i are distinct. This means that $|\mathscr{C}| \leqq V(n, 3) - (n+1)$, where $n = |Q|$.

LEMMA 4.2. *For* $t = 2$ *and* $t = 3$, *the* t-ID *Problem is* NP-*complete.*

Proof. First, it is easy to verify that the t-ID Problem is in NP. Now, we use the following reduction from the Hyper-t-Col Problem to the t-ID Problem. Given an instance $(Q = \{q_0, q_1, \ldots, q_{n-1}\}, \mathscr{C})$ of the Hyper-t-Col Problem, let $\mathbf{u}_i = [u_{i,0}\, u_{i,1} \ldots u_{i,n-1}] \in \{0, 1\}^n$ be the characteristic vector of $Q - Q_i$, $1 \leqq i \leqq |\mathscr{C}|$. That is, $u_{i,j} = 0$ if and only if $q_j \in Q_i$. Also, let $\mathbf{e}_j$ denote the vector in $\{0, 1\}^n$ of weight $n-1$ whose zero is at location j, $0 \leqq j \leqq n-1$. The corresponding instance of the t-ID Problem

is now given by $(n, R_{(Q,\mathscr{C})})$, where

$$R_{(Q,\mathscr{C})} \triangleq \{(\mathbf{e}_j\,;\,1)\}_{j=0}^{n-1} \cup \{(\mathbf{u}_i\,;\,0)\}_{i=1}^{|\mathscr{C}|}.$$

Note that $|R_{(Q,\mathscr{C})}| < V(n,3)$.

The proof of the validity of the reduction is very similar to the proof in [11], and it is presented here for the sake of completeness. First, assume that $(Q = \{q_0, q_1, \ldots, q_{n-1}\}, \mathscr{C})$ is t-colorable by a coloring $\chi: Q \to \{1, 2, \ldots, t\}$. We show the existence of a t-sparse function $f: \{0,1\}^n \to \{0,1\}$ which passes via the set $R_{(Q,\mathscr{C})}$. Let f be the n-variable polynomial defined by $f = \sum_{l=1}^{t} T_l$, where each T_l is a monomial given by

$$T_l = \prod_{j \text{ s.t. } \chi(q_j) \neq l} x_j,$$

and $T_l \triangleq 1$ if all the q_j are colored by l (in which case $\mathscr{C}$ must be empty). Clearly, each variable x_j is missing from exactly one monomial $(T_{\chi(q_j)})$ and, therefore, $T_l(\mathbf{e}_j) = \delta(l, \chi(q_j))$, where $\delta(\cdot,\cdot)$ stands for the Kronecker delta function. We thus have $f(\mathbf{e}_j) = 1$ for all $0 \leqq j \leqq n-1$. Now, let $Q_i \in \mathscr{C}$, let $\mathbf{u}_i = [u_{i,0}\ u_{i,1} \ldots u_{i,n-1}]$ be the characteristic vector of $Q - Q_i$ and suppose, to the contrary, that $f(\mathbf{u}_i) = 1$. This implies $T_l(\mathbf{u}_i) = 1$ for at least one l and, therefore, we must have $u_{i,j} = 1$ for every j such that $\chi(q_j) \neq l$. Hence, whenever $u_{i,j} = 0$ we have $\chi(q_j) = l$, implying that all the elements of Q_i have the same color—a contradiction.

We now show that the existence of an n-variable t-sparse function f satisfying $f(\mathbf{v}) = s$ for every $(\mathbf{v}; s) \in R_{(Q,\mathscr{C})}$ implies the existence of a valid coloring of Q. Suppose that such a function f exists, and write $f = \sum_{l=1}^{k} T_l$, $k \leqq t$ $(\leqq 3)$, where each T_l is a nonzero monomial in the variables x_j, $0 \leqq j \leqq n-1$. First, we show that if x_j appears in f at least once, then it must be missing from exactly one monomial. Indeed, assume that x_j appears in T_1, implying $T_1(\mathbf{e}_j) = 0$. Since $f(\mathbf{e}_j) = \sum_{l=1}^{k} T_l(\mathbf{e}_j) = 1$, we must have $T_l(\mathbf{e}_j) = 1$ for exactly one monomial T_l, $2 \leqq l \leqq k$, the only monomial from which x_j is absent. Therefore, every variable x_j which appears in f can be assigned a well-defined index $l = l(x_j)$ of the monomial T_l from which it is missing.

Now, define a coloring $\chi: Q \to \{1, 2, \ldots, t\}$ as follows. If x_j appears in f, then $\chi(q_j) = l(x_j)$; otherwise, assign $\chi(q_j) = 1$. We now show that the above is indeed a valid coloring of Q. Suppose, to the contrary, that there exists a constraint $Q_i \in \mathscr{C}$ such that $\chi(q_j) = l_0$ for all $q_j \in Q_i$. Assume first that, for some $q_j \in Q_i$, the corresponding x_j appears in f (in which case $l_0 = l(x_j)$), and let $\mathbf{u}_i$ be the characteristic vector of $Q - Q_i$. Since x_j appears in each T_l, $l \neq l_0$, we have $f(\mathbf{u}_i) = T_{l_0}(\mathbf{u}_i) = 0$. This means that for at least one variable x_r appearing in T_{l_0}, the corresponding entry $u_{i,r}$ in $\mathbf{u}_i$ must be zero, implying $q_r \in Q_i$. On the other hand, $l(x_r) \neq l_0$ and, therefore, $\chi(q_r) \neq l_0$, contradicting the assumption that all elements of Q_i are colored by l_0.

It remains to consider the case where there exists a constraint $Q_i \in \mathscr{C}$ such that neither of the variables x_j, corresponding to $q_j \in Q_i$, appear in f. Now, since $f(\mathbf{u}_i) = 0$, f must have an even number of nonzero monomials. Hence, for every $q_j \in Q_i$, the corresponding vector $\mathbf{e}_j$ satisfies $f(\mathbf{e}_j) = 0$, resulting in a contradiction. $\square$

Proof of Theorem 4.1. To complete the proof of the theorem we present a reduction from the t-ID Problem to the $(t+2)$-ID Problem. Let (n, R_t) be an instance of the t-ID Problem; the corresponding instance of the $(t+2)$-ID Problem is given by $(n+2, R_{t+2})$, where

$$\begin{aligned} R_{t+2} = &\{(11\mathbf{v}\,;\,s) \mid (\mathbf{v}\,;\,s) \in R_t\} \\ &\cup \{(0\,0 \ldots 0;\,0)\} \\ &\cup \{(0\,1\,0\,0 \ldots 0;\,1), (1\,0\,0\,0 \ldots 0;\,1)\} \end{aligned}$$

(the size of R_{t+2} is, therefore, $|R_t| + 3$).

To prove the validity of the reduction, we must show that there exists an n-variable t-sparse function f_t satisfying R_t if and only if there exists an $(n+2)$-variable $(t+2)$-sparse function f_{t+2} satisfying R_{t+2}. Indeed, if f_t satisfies R_t, then

$$f_{t+2}(x_{n+1}, x_n, x_{n-1}, \ldots, x_0) \triangleq x_{n+1}x_n \cdot f_t(x_{n-1}, \ldots, x_0) + x_{n+1} + x_n$$

satisfies R_{t+2}.

On the other hand, let f_{t+2} be an $(n+2)$-variable $(t+2)$-sparse function satisfying R_{t+2}. Since $f_{t+2}(0, 0, \ldots, 0) = 0$ and $f_{t+2}(0, 1, 0, 0, \ldots, 0) = f_{t+2}(1, 0, 0, 0, \ldots, 0) = 1$, we can write $f_{t+2} = \phi + x_{n+1} + x_n$, where ϕ is an $(n+2)$-variable t-sparse function containing neither the linear terms x_{n+1} and x_n, nor the constant 1. We now define the t-sparse function $f_t : \{0, 1\}^n \to \{0, 1\}$ by $f_t(\mathbf{x}) \triangleq \phi(11\mathbf{x})$. Since $f_{t+2}(11\mathbf{v}) = f_t(\mathbf{v})$ for every $\mathbf{v} \in \{0, 1\}^n$, f_t must satisfy R_t. □

By the proofs of Lemma 4.2 and Theorem 4.1 it follows that Theorem 4.1 still holds even if we restrict the size of $R = \{(\mathbf{v}_i; s_i)\}_i$ to be smaller than $V(n, 3)$. When $S(n, 1 + \lfloor \log_2 t \rfloor)$ is contained in R, however, the t-Interpolation Decision Problem is easy to solve.

5. Approximation of Boolean functions. In the following sections we consider the problem of approximating t-sparse functions $f : \{0, 1\}^n \to \{0, 1\}$, given the values of f at various evaluation points in $\{0, 1\}^n$. Let $\mathbf{F}$ and $\mathbf{G}$ be the truth tables of two functions $f, g : \{0, 1\}^n \to \{0, 1\}$, and let ε be a real number in the interval $(0, 1]$. We say that f and g are *ε-close* if $w(\mathbf{F} + \mathbf{G}) < \varepsilon \cdot 2^n$, or, equivalently, if $\text{Prob}[f(\mathbf{x}) \neq g(\mathbf{x})] < \varepsilon$, where $\mathbf{x}$ is chosen uniformly from the space $\{0, 1\}^n$. Given a function f, any function g which is ε-close to f will serve as an *ε-approximation* of f. Two functions which are not ε-close are said to be *ε-far.*

A set P of points in $\{0, 1\}^n$ is called an *ε-approximation set* for t-sparse functions, if every two t-sparse functions $f, g : \{0, 1\}^n \to \{0, 1\}$, taking the same values at P, are necessarily ε-close. Having such a set P and the values of a t-sparse function f at P, we can ε-approximate f by taking any t-sparse function g whose truth table coincides with that of f at P. On the other hand, consider a set Q of points in $\{0, 1\}^n$ such that knowing the values of *any* t-sparse function f at Q is sufficient for finding an $(\varepsilon/2)$-approximating function $\hat{f}$ for f. In such a case, Q must be an ε-approximation set, since every two t-sparse functions f and g whose truth tables coincide at Q have the same $(\varepsilon/2)$-approximating functions $\hat{f}$. By the triangle inequality f and g must therefore be ε-close.

We begin by obtaining bounds on the minimum size $L(n, t, \varepsilon)$ of any ε-approximation set for t-sparse functions over $\{0, 1\}^n$ (note that our discussion in the foregoing sections corresponds to the special case $\varepsilon \leqq 2^{-n}$). As in the interpolation case, we shall concentrate on the nonadaptive model, pointing out that similar bounds can be obtained for the adaptive case as well.

Let $f : \{0, 1\}^n \to \{0, 1\}$ be a t-sparse function with each monomial being a product of at least k variables. Then, it is easy to see that the truth table of f is of Hamming weight $\leqq t \cdot 2^{n-k}$. On the other hand, by the properties of Reed–Muller codes we have the following lemma.

LEMMA 5.1. *Let the polynomial representation of a nonzero function $f : \{0, 1\}^n \to \{0, 1\}$ consist of a sum of monomials, each being a product of at most k variables. Then, the truth table* $\mathbf{F}$ *of f satisfies*

$$w(\mathbf{F}) \geqq 2^{n-k}.$$

Proof. The vector $\mathbf{F}$ is a nontrivial linear combination of columns of A_n (cf. § 2) whose indices are of Hamming weight $\leqq k$. Now, these column vectors are exactly the rows of $H_{n,k}$. Therefore, $\mathbf{F}$ is a nonzero codeword of the kth order Reed-Muller code of length 2^n (which is the *dual* of the $(n-k-1)$st order Reed-Muller code [9, p. 376]) and, as such, its weight is at least 2^{n-k}. □

Let $\Gamma(n, t, k)$ denote the set of all t-sparse functions $f:\{0,1\}^n \to \{0,1\}$ such that each monomial in the polynomial representation of f is a product of at most k variables. Note that if f and g are two distinct functions in $\Gamma(n, t, k)$, then $f-g \in \Gamma(n, 2t, k)-\{0\}$ and, therefore, by Lemma 5.1, f and g are 2^{-k}-far. Hence, any two such functions should not take the same values at any ε-approximation set. Setting $k = \lfloor -\log_2 \varepsilon \rfloor$, we obtain the following information bound

$$L(n, t, \varepsilon) \geqq L(n, t, 2^{-k}) \geqq \log_2 |\Gamma(n, t, k)| = \log_2 V(V(n, \lfloor -\log_2 \varepsilon \rfloor), t). \tag{7}$$

For a large range of values of n, t, and ε, we can obtain a tighter lower bound on $L(n, t, \varepsilon)$ which is presented in Theorem 5.1, following the next definitions.

Let $H:[0,1] \to [0,1]$ be the function given by

$$H(x) = \begin{cases} 0 & \text{if } x = 0 \\ -x \cdot \log_2 x - (1-x) \cdot \log_2 (1-x) & \text{if } 0 < x \leqq \frac{1}{2}, \\ 1 & \text{otherwise} \end{cases}$$

and, for $0 \leqq \rho \leqq 1$, let $E(\rho)$ be the curve in the real plane defined by

$$E(\rho) \triangleq \{(\delta, \mu) \mid H((1-\mu)\delta) = \rho \cdot (1 + \delta H(\mu)), 0 \leqq \delta \leqq 1, 0 \leqq \mu \leqq \tfrac{1}{2}\}.$$

Using the notations $\rho_1 \triangleq ((5-\sqrt{5})/10) \approx 0.276$ and $\rho_2 \triangleq ((5+\sqrt{5})/10) \cdot H((3-\sqrt{5})/4) \approx 0.509$, we now define the function $\gamma:[0,1] \to [0,1]$ by

$$\gamma(\rho) = \begin{cases} (1-\rho) \cdot H(\rho/(1-\rho)) & \text{if } 0 \leqq \rho \leqq \rho_1 \\ \log_2((1+\sqrt{5})/2)\,(\approx 0.694) & \text{if } \rho_1 < \rho \leqq \rho_2. \\ \max_{(\delta,\mu) \in E(\rho)} \{H(\delta)/(1+\delta H(\mu))\} & \text{if } \rho_2 < \rho \leqq 1 \end{cases} \tag{8}$$

Fig. 2 depicts $\gamma(\rho)$ versus ρ. Some of the properties of $\gamma(\cdot)$ are summarized in Lemma 5.5 and Remarks 5.1-5.3 below.

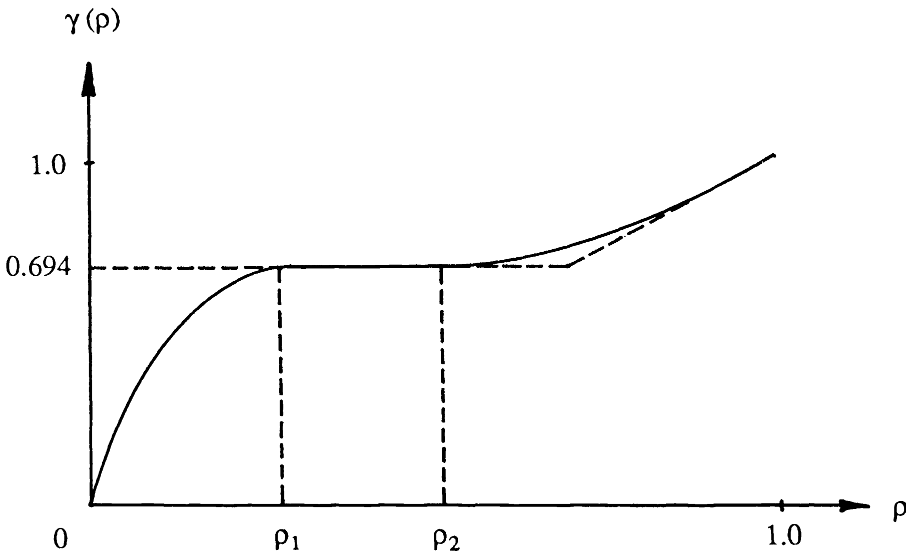


FIG. 2. $\gamma(\rho)$ *versus* ρ.

THEOREM 5.1. *For* $t>0$ *and* $\varepsilon \in (0, 1]$, *let* $\alpha(t, \varepsilon) \triangleq \gamma((\log t)/(\log (t/\varepsilon)))$. *Given any fixed* $\theta > 0$, *there exists an integer* τ, *depending only on* θ, *such that*

$$L(n, t, \varepsilon) = \Omega\left((t/\varepsilon)^{\alpha(t,\varepsilon)-\theta} \cdot \log\left(\frac{1}{n+2-\log_2 (t/\varepsilon)} + \sqrt[\tau]{4\varepsilon}\right)^{-1}\right) \tag{9}$$

whenever $\varepsilon < \frac{1}{4}$ and $\tau \leqq t/\varepsilon \leqq 2^n$.

Here $\Omega(g(n))$ stands for an expression which is bounded from below by $a \cdot g(n)$ for some positive constant a. When $t/\varepsilon \ll 2^n$ and $\varepsilon \leqq 1/n^c$ (for some fixed positive constant c), (9) becomes

$$L(n, t, \varepsilon) = \Omega((t/\varepsilon)^{\alpha(t,\varepsilon)-\theta} \cdot \log n).$$

On the other hand, when ε is constant (independent of n) the information bound (7) yields a better bound than (9). Note also that the range $t/\varepsilon \geqq 2^n$ has been excluded from Theorem 5.1 (see, however, Remark 5.4 below); in fact, in this range of parameters the values of the underlying function f can be specified at all points of $S(n, 1+\lfloor \log_2 t \rfloor)$ (Theorem 2.1), still obtaining an algorithm whose time complexity is polynomial in t and $1/\varepsilon$. As the proof of Theorem 5.1 is rather long, we postpone it to the end of this section.

The following is the analog of Theorem 5.1 for the t-term DNF case. Let $L_{\mathrm{DNF}}(n, t, \varepsilon)$ denote the minimum size of any DNF ε-approximation set, i.e., a set $P \subseteq \{0, 1\}^n$ such that every two t-term DNF functions over $\{0, 1\}^n$, taking the same values at P, are necessarily ε-close.

THEOREM 5.2. (i) *For* $\varepsilon < \frac{1}{4}$ *and* $t/\varepsilon < 2^{n-1}$,

$$L_{\mathrm{DNF}}(n, t, \varepsilon) \geqq \frac{1}{8} \cdot (t/\varepsilon) \cdot \log_2\left(\frac{1}{n+1-\log_2 (t/\varepsilon)} + 4\varepsilon\right)^{-1}.$$

(ii) *For* $\varepsilon < \frac{1}{2}$ *and* $t/\varepsilon \geqq 2^{n-1}$,

$$L_{\mathrm{DNF}}(n, t, \varepsilon) = \Omega(2^n).$$

The proof of Theorem 5.2 is presented after that of Theorem 5.1.[1]

The following theorem establishes a nonconstructive upper bound on $L(n, t, \varepsilon)$.

THEOREM 5.3. *Given* n, t, *and* $0 < \varepsilon < 1$,

$$L(n, t, \varepsilon) \leqq \left\lceil \frac{\log V(2^n, 2t)}{-\log (1-\varepsilon)} \right\rceil \tag{10}$$

and, therefore,

$$L(n, t, \varepsilon) = O((t/\varepsilon) \cdot n).$$

Proof. Let K denote the set of all $2t$-sparse functions f over $\{0, 1\}^n$ with $w(\mathbf{F}) \geqq \varepsilon \cdot 2^n$. It is sufficient to show that if L is an integer not smaller than the right-hand side of (10), then there exists an $L \times 2^n$ submatrix H of A_n (cf. § 2) such that for any $f \in K$ we have $H\mathbf{f} \neq \mathbf{0}$.

For every $f \in K$ there exist less than $(1-\varepsilon) \cdot 2^n$ rows $\mathbf{a}$ in A_n for which $\mathbf{a} \cdot \mathbf{f} = 0$. Therefore, for every integer L there exist less than $(1-\varepsilon)^L \cdot 2^{nL} \cdot |K|$ distinct $L \times 2^n$ matrices H, with rows taken from A_n, such that $H\mathbf{f} = \mathbf{0}$ for at least one f in K (here H may contain the same row of A_n more than once). Now, $|K| \leqq V(2^n, 2t)$ and, so, if

$$(1-\varepsilon)^L \cdot 2^{nL} \cdot V(2^n, 2t) \leqq 2^{nL}, \tag{11}$$

[1] Due to integer roundings, the proofs of Theorems 5.1 and 5.2 yield slightly better lower bounds than the ones stated in the above theorems. However, for the sake of clarity, we chose to state these theorems in their present form.

we can always find an $L \times 2^n$ matrix H for which $H\mathbf{f} \neq \mathbf{0}$ whenever $f \in K$. The theorem now follows by taking the logarithms of both sides of (11). $\square$

The t-term DNF analog of Theorem 5.3 takes the form of the following theorem.

THEOREM 5.4. *Given n, t, and $0 < \varepsilon < 1$,*

$$L_{\mathrm{DNF}}(n, t, \varepsilon) \leqq \left\lceil \frac{2 \cdot \log\left(\sum_{i=0}^{t} 2^i \cdot \binom{2^n}{i}\right)}{-\log(1-\varepsilon)} \right\rceil = O((t/\varepsilon) \cdot n).$$

Proof. The proof here is similar to that of Theorem 5.3. Let Γ_{DNF} denote the set of all t-term DNF functions over $\{0, 1\}^n$. Given an integer L and two functions f_1, $f_2 \in \Gamma_{\mathrm{DNF}}$ which are ε-far, there exist less than $(1-\varepsilon)^L \cdot 2^{nL}$ ordered multisets P of L points in $\{0, 1\}^n$ such that both f_1 and f_2 take the same values at P. Hence, if

$$(1-\varepsilon)^L \cdot 2^{nL} \cdot |\Gamma_{\mathrm{DNF}}|^2 \leqq 2^{nL},$$

we can always find a DNF ε-approximation set of size $\leqq L$. The theorem now follows by the inequality $|\Gamma_{\mathrm{DNF}}| \leqq \sum_{i=0}^{t} 2^i \cdot \binom{2^n}{i}$. $\square$

We now turn to the proof of Theorem 5.1, starting with a series of lemmas.

LEMMA 5.2. *Given integers n, m, r, and s, where $2 \leqq m \leqq n$ and $0 \leqq r \leqq m$, let B be an $L \times n$ binary matrix such that every $L \times m$ submatrix of B contains among its rows at least $V(m, r) - s$ distinct elements of $S(m, r)$. Then,*

$$L \geqq V(m-2, r-1) \cdot \left(\log_2(n-m+2) - \log_2\left(1 + \frac{s(n-m+1)}{2V(m-2, r-1)}\right)\right).$$

A special case of this combinatorial result, for $r = m$ and $s = 0$, was proved in [12].

Proof. For every $\mathbf{u} \in S(m-2, r-1)$, let $L_{\mathbf{u}}$ denote the number of rows of B whose $(m-2)$-suffix is equal to $\mathbf{u}$, and let $C_{\mathbf{u}}$ denote the $L_{\mathbf{u}} \times (n-m+2)$ submatrix of B consisting of the $(n-m+2)$-prefixes of these rows (in case $L_{\mathbf{u}} = 0$, $C_{\mathbf{u}}$ denotes an "empty" matrix). Assuming that $L_{\mathbf{u}} > 0$, let $M_{\mathbf{u}}$ denote the number of pairs of identical columns in $C_{\mathbf{u}}$, and let N denote the number of distinct columns in $C_{\mathbf{u}}$, each such column appearing n_i times in $C_{\mathbf{u}}$, $i = 1, 2, \ldots, N$. We have

$$\sum_{i=1}^{N} n_i = n - m + 2, \tag{12}$$

and

$$2 \cdot M_{\mathbf{u}} = 2 \cdot \sum_{i=1}^{N} \binom{n_i}{2} = \sum_{i=1}^{N} n_i^2 - \sum_{i=1}^{N} n_i.$$

Since $N \cdot \sum_{i=1}^{N} n_i^2 \geqq \left(\sum_{i=1}^{N} n_i\right)^2$ we thus obtain

$$2 \cdot M_{\mathbf{u}} \geqq \frac{1}{N} \cdot \left(\sum_{i=1}^{N} n_i\right)^2 - \sum_{i=1}^{N} n_i \stackrel{(12)}{=} \frac{(n-m+2)^2}{N} - (n-m+2),$$

or

$$N \geqq \frac{(n-m+2)^2}{2M_{\mathbf{u}} + n - m + 2}.$$

On the other hand, we must have $L_{\mathbf{u}} \geqq \log_2 N$ to allow N distinct columns in $C_{\mathbf{u}}$. Therefore,

$$L_{\mathbf{u}} \geqq \log_2\left(\frac{(n-m+2)^2}{2M_{\mathbf{u}} + n - m + 2}\right),$$

yielding

$$(13)\quad \begin{aligned} L &\geqq \sum_{\mathbf{u}\in S(m-2,r-1)} L_{\mathbf{u}} \geqq \sum_{\mathbf{u}\in S(m-2,r-1)} \log_2\left(\frac{(n-m+2)^2}{2M_{\mathbf{u}}+n-m+2}\right) \\ &= 2\cdot V(m-2,r-1)\cdot \log_2(n-m+2) - \sum_{\mathbf{u}\in S(m-2,r-1)} \log_2(2M_{\mathbf{u}}+n-m+2). \end{aligned}$$

Let $\mathbf{c}_i$ and $\mathbf{c}_j$ be two identical columns (if any) in $C_{\mathbf{u}}$. These two columns define two row vectors, namely, $[0\,1\,\mathbf{u}]$ and $[1\,0\,\mathbf{u}]$, both in $S(m, r)$, which are *missing* from the $L\times m$ submatrix of B consisting of the ith and jth columns, together with the last $m-2$ columns of B. Enumerating over all pairs of columns out of the first $n-m+2$ columns of B, we obtain

$$(14)\qquad 2\cdot \sum_{\mathbf{u}\in S(m-2,r-1)} M_{\mathbf{u}} \leqq s\cdot \binom{n-m+2}{2}.$$

Since the logarithmic function is convex, we can use Jensen's inequality [10, p. 277] to obtain

$$(15)\quad \begin{aligned} &\frac{\sum_{\mathbf{u}\in S(m-2,r-1)} \log_2(2M_{\mathbf{u}}+n-m+2)}{V(m-2,r-1)} \\ &\leqq \log_2\left(\frac{\sum_{\mathbf{u}\in S(m-2,r-1)}(2M_{\mathbf{u}}+n-m+2)}{V(m-2,r-1)}\right) \\ &= \log_2\left((n-m+2)+\frac{2\sum_{\mathbf{u}\in S(m-2,r-1)} M_{\mathbf{u}}}{V(m-2,r-1)}\right) \\ &\overset{(14)}{\leqq} \log_2\left((n-m+2)+\frac{s\cdot\binom{n-m+2}{2}}{V(m-2,r-1)}\right) \\ &= \log_2(n-m+2)+\log_2\left(1+\frac{s\cdot(n-m+1)}{2V(m-2,r-1)}\right). \end{aligned}$$

Combining (13) and (15) we thus obtain

$$\begin{aligned} L &\geqq 2\cdot V(m-2,r-1)\cdot\log_2(n-m+2) \\ &\quad - V(m-2,r-1)\left(\log_2(n-m+2)+\log_2\left(1+\frac{s\cdot(n-m+1)}{2V(m-2,r-1)}\right)\right) \\ &= V(m-2,r-1)\left(\log_2(n-m+2)-\log_2\left(1+\frac{s\cdot(n-m+1)}{2V(m-2,r-1)}\right)\right). \end{aligned}$$ □

LEMMA 5.3. *Given n, t, and $2^{-n}\leqq\varepsilon\leqq 1$, let $k\triangleq\lfloor-\log_2\varepsilon\rfloor$ and let m and r be integers satisfying the following two conditions*:

(i) $\max(k,2)\leqq m\leqq n$; *and*

(ii) *there exists an integer l, $0\leqq l\leqq r$, such that $2^{m-k}\cdot V(r,l)+V(m,r-l-1)\leqq 2t$.*

Then,

$$L(n,t,\varepsilon)\geqq V(m-2,r-1)\cdot\left(\log_2(n-m+2)-\log_2\left(1+\frac{(2^{m-k}-1)(n-m+1)}{2V(m-2,r-1)}\right)\right).$$

Proof. Let $L\triangleq L(n,t,\varepsilon)$ and let B be an $L\times n$ binary matrix whose rows form an ε-approximation set of size L. Let m and r be integers satisfying conditions (i) and (ii), and let C be an $L\times m$ submatrix of B consisting, say, of the last m columns of B. We now claim that C contains among its rows at least $V(m,r)-2^{m-k}+1$ distinct row vectors of $S(m,r)$.

Assume, to the contrary, that 2^{m-k} such rows are missing from C, say the rows $\mathbf{z}_i=[z_{i,m-1}z_{i,m-2}\dots z_{i,0}]$, $1\leqq i\leqq 2^{m-k}$. For each $\mathbf{z}_i$ we associate a term $\phi_i=\prod_{j=0}^{m-1} y_{i,j}$, where $y_{i,j}=x_j$ if $z_{i,j}=1$, and $y_{i,j}=\bar{x}_j$ otherwise. It is easy to verify that for every $\mathbf{u}\in\{0,1\}^m$, $\phi_i(\mathbf{u})=1$ if and only if $\mathbf{u}=\mathbf{z}_i$. Substituting $1+x_j$ for $\bar{x}_j$ in ϕ_i, and expanding the expressions thus obtained, each ϕ_i becomes a sum of up to 2^r monomials.

For each i, $1\leqq i\leqq 2^{m-k}$, let ξ_i denote the sum of those monomials in ϕ_i which are products of at most $m-r+l$ variables, where l is an integer guaranteed by condition (ii). Write $\phi\triangleq\sum_i\phi_i$ and $\xi\triangleq\sum_i\xi_i$ and let $\eta\triangleq\phi-\xi$. It can be verified that ξ is a sum of up to $2^{m-k}\cdot V(r,l)$ monomials and η is a sum of up to $V(m,r-l-1)$ monomials. Hence, by condition (ii), ϕ is $2t$-sparse. On the other hand, the truth table of ϕ, when regarded as an n-variable function, is of weight $2^{m-k}\cdot 2^{n-m}\geqq\varepsilon\cdot 2^n$ and so, ϕ can be written as a sum of two functions which are both t-sparse and ε-far, contradicting the fact that they take the same values at an ε-approximation set. Since the above discussion applies to any $L\times m$ submatrix C of B, we can now apply Lemma 5.2 with $s=2^{m-k}-1$, thus concluding the proof of the lemma. □

When $k=n$ we can set $m=n$ and $r=l=1+\lfloor\log_2 t\rfloor$ in Lemma 5.3, yielding $L(n,t,2^{-n})\geqq V(n-2,\lfloor\log_2 t\rfloor)$, which, in view of Theorem 2.1, is quite close to the true value. Theorem 5.1 is virtually a restatement of Lemma 5.3, optimizing with respect to m, r, and l, and using the following well-known approximation of $V(n,r)$ (see, for instance, [9, p. 310]).

LEMMA 5.4. *For every two integers n and $r=\mu\cdot n$, $0\leqq\mu\leqq 1$,*

$$n\cdot H(\mu)-\tfrac{1}{2}\log_2(2n)\leqq\log_2 V(n,r)\leqq n\cdot H(\mu).$$

LEMMA 5.5. *Let ψ, ω_1, ω_2: $[0,1]\times[0,1]\to[0,1]$ be given by*

$$\psi(\delta,\mu)\triangleq\frac{H(\delta)}{1+\delta H(\mu)};$$

$$\omega_1(\delta,\mu)\triangleq\frac{\delta H(\mu)}{1+\delta H(\mu)};$$

and

$$\omega_2(\delta,\mu)\triangleq\frac{H((1-\mu)\delta)}{1+\delta H(\mu)}.$$

For any $\rho\in[0,1]$, let $D_1(\rho)$ and $D_2(\rho)$ be the sets of pairs (δ,μ) in the unit square $[0,1]\times[0,1]$ defined by

$$D_i(\rho)=\{(\delta,\mu)\in[0,1]\times[0,1]\mid\rho\geqq\omega_i(\delta,\mu)\},\qquad i=1,2, \tag{16}$$

and let γ^*: $[0,1]\to[0,1]$ be defined by

$$\gamma^*(\rho)\triangleq\max_{(\delta,\mu)\in D_1(\rho)\cap D_2(\rho)}\{\psi(\delta,\mu)\}. \tag{17}$$

Then,

$$\gamma^*(\rho)\geqq\gamma(\rho),\qquad\rho\in[0,1],$$

where $\gamma(\cdot)$ is defined by (8).

Proof. Let $X_1(\rho)$ and $X_2(\rho)$ be the sets given by

$$X_1(\rho)\triangleq\{(\delta,\mu)\in D_1(\rho)\cap D_2(\rho)\mid\mu\geqq\tfrac{1}{2}\}$$

and

$$X_2(\rho)\triangleq\{(\delta,\mu)\in D_1(\rho)\cap D_2(\rho)\mid\mu\leqq\tfrac{1}{2}\}, \tag{18}$$

and let the functions $\gamma_1, \gamma_2 : [0,1] \to [0,1]$ be defined by

$$\gamma_i(\rho) \triangleq \max_{(\delta,\mu)\in X_i(\rho)} \{\psi(\delta, \mu)\}, \qquad i = 1, 2. \tag{19}$$

Clearly, $\gamma^*(\rho) = \max\{\gamma_1(\rho), \gamma_2(\rho)\}$.

We start by analyzing the function $\gamma_1(\cdot)$. Given $\rho \in [0, 1]$, $X_1(\rho)$ is equal to the set of pairs $(\delta, \mu) \in [0, 1] \times [\frac{1}{2}, 1]$ satisfying both

$$\rho \geqq \omega_1(\delta, \mu) = \omega_1(\delta, 1) = \frac{\delta}{1+\delta} \tag{20}$$

and

$$\rho \geqq \omega_2(\delta, \mu). \tag{21}$$

Note that (20) is independent of μ, as is the expression $\psi(\delta, \mu) = \psi(\delta, 1)$, which is to be maximized in (19) to obtain $\gamma_1(\rho)$. Also, (21) is satisfied for every ρ and δ if $\mu = 1$. Therefore, by (19) and (20) we can write

$$\gamma_1(\rho) = \max_{0\leqq\delta\leqq\min\{1,\rho/(1-\rho)\}} \{\psi(\delta, 1)\}. \tag{22}$$

The maximum value of $\psi(\cdot, 1)$ in the interval $[0, 1]$ is attained at $\delta_0 \triangleq ((3-\sqrt{5})/2)$, in which case $\psi(\delta_0, 1) = \log_2((1+\sqrt{5})/2) \approx 0.694$. Hence, for $\rho \geqq \rho_1 = \omega_1(\delta_0, 1) = \delta_0/(1+\delta_0) = ((5-\sqrt{5})/10) \approx 0.276$, we have $\gamma_1(\rho) = \psi(\delta_0, 1) = \log_2((1+\sqrt{5})/2)$. Since $\psi(\delta, 1)$ is monotonously increasing when $\delta < \delta_0$, the maximum in (22) for $\rho \leqq \rho_1$ is attained when $\delta = \rho/(1-\rho)$. Hence,

$$\gamma_1(\rho) = \begin{cases} (1-\rho)\cdot H(\rho/(1-\rho)) & \text{if } 0 \leqq \rho \leqq \rho_1 \\ \log_2((1+\sqrt{5})/2) & \text{if } \rho_1 < \rho \leqq 1 \end{cases},$$

implying $\gamma_1(\rho) = \gamma(\rho)$ for $0 \leqq \rho \leqq \rho_2$.

We now turn to the function $\gamma_2(\cdot)$. For every $\delta \in [0, 1]$ and $\mu \leqq \frac{1}{2}$, we have $\delta H(\mu) \leqq \delta \leqq H(\delta/2) \leqq H((1-\mu)\delta)$ and, therefore, (18) boils down to

$$X_2(\rho) = \{(\delta, \mu) \in [0, 1] \times [0, \tfrac{1}{2}] \mid \rho \geqq \omega_2(\delta, \mu)\},$$

implying $\gamma_2(\rho) \geqq \gamma(\rho)$ for $\rho_2 < \rho \leqq 1$. Therefore, $\gamma^*(\rho) \geqq \gamma(\rho)$ for every $\rho \in [0, 1]$. □

Remark 5.1. Referring to the notations of the last proof, we can verify that the functions γ_1 and γ_2, and therefore γ and γ^*, are all nondecreasing. Indeed, for any $\rho \leqq \hat{\rho}$ we have $X_i(\rho) \subseteq X_i(\hat{\rho})$, $i = 1, 2$.

Remark 5.2. For fixed δ, both $\psi(\delta, \mu)$ and $\omega_2(\delta, \mu)$ are monotonously nonincreasing with respect to μ, whereas $\omega_1(\delta, \mu)$ is monotonously nondecreasing. Hence, if $(\delta(\rho), \mu(\rho))$ is a pair attaining the maximum in (17) for a given ρ, we can assume that $\delta(\rho)$ and $\mu(\rho)$ are such that $\rho = \omega_2(\delta(\rho), \mu(\rho))$. We thus have

$$\frac{\gamma^*(\rho)}{\rho} = \frac{\psi(\delta(\rho), \mu(\rho))}{\omega_2(\delta(\rho), \mu(\rho))} = \frac{H(\delta(\rho))}{H((1-\mu(\rho))\delta(\rho))} \geqq 1; \tag{23}$$

that is, $\gamma^*(\rho)$ is always above (or on) the line $\rho \mapsto \rho$. Furthermore, it can be readily verified that both $\delta(\rho)$ and $\mu(\rho)$ are nonzero, unless $\rho \in \{0, 1\}$, implying a strict inequality in (23) whenever $\rho \in (0, 1)$.

Remark 5.3. Similarly,

$$\gamma_2(\rho) = \max_{\rho\geqq\omega_2(\delta,\mu)} \psi(\delta, \mu) = \max_{\rho=\omega_2(\delta,\mu)} \psi(\delta, \mu), \tag{24}$$

i.e., $\gamma_2(\rho)=\gamma(\rho)$ whenever $\rho_2<\rho\leqq 1$. Note also that

$$\rho_2=\omega_2\left(\delta_0,\frac{1}{2}\right)=\frac{5+\sqrt{5}}{10}\cdot H\left(\frac{3-\sqrt{5}}{4}\right)\approx 0.509$$

and that

$$\gamma_2(\rho_2)\geqq\frac{H(\delta_0)}{1+\delta_0 H(\frac{1}{2})}=\log_2\left(\frac{1+\sqrt{5}}{2}\right). \tag{25}$$

In fact, by applying Lagrange multipliers on (24), it can be verified that (25) holds with equality; this implies that $\gamma(\cdot)$ is continuous (even differentiable) within the interval $[0,1]$ and that $\gamma^*(\rho)$ is actually *equal* to $\gamma(\rho)$.

Proof of Theorem 5.1. Given n, t, and ε, let $h\triangleq\lfloor\log_2 t\rfloor$, $k\triangleq\lfloor-\log_2\varepsilon\rfloor$ ($\geqq 2$) and $\rho\triangleq h/(k+h)$. By the continuity of $\gamma(\rho)$, for every $\theta>0$ there exists an integer $N_0(\theta)$ such that

$$\alpha(t,\varepsilon)=\gamma\left(\frac{\log t}{\log(t/\varepsilon)}\right)\leqq\gamma\left(\rho+\frac{1}{k+h}\right)\leqq\gamma(\rho)+\theta/2$$

whenever $k+h\geqq N_0(\theta)$. Therefore, we choose τ to be at least $2^{N_0(\theta)+1}$, allowing us to replace the exponent $\alpha(t,\varepsilon)-\theta$ in (9) by $\gamma(\rho)-\theta/2$, thus simplifying the analysis in the sequel.

Given n, k, and h, let m, r, and l be integers satisfying the following three conditions:

(a) $k\leqq m\leqq n$;
(b) $2^{m-k}\cdot V(r,l)\leqq 2^h$; and
(c) $V(m,r-l-1)\leqq 2^h$.

Using the notation $\sigma(m,k,r)\triangleq 2^{m-k-1}/V(m-2,r-1)$, by Lemma 5.3 we have

$$L(n,t,\varepsilon)\geqq V(m-2,r-1)\cdot\log_2\left(\frac{1}{n-m+2}+\sigma(m,k,r)\right)^{-1} \tag{26}$$

for any m, r, and l satisfying (a)-(c). We now maximize $V(m-2,r-1)$ under the above three conditions.

Let $\mu\triangleq l/r>0$. By Lemma 5.4, (b) is implied by $2^{m-k}\cdot 2^{rH(\mu)}\leqq 2^h$ and so, we can set

$$r=\left\lfloor\frac{k+h-m}{H(\mu)}\right\rfloor. \tag{27}$$

Let m be in the range $\frac{1}{2}(k+h)\leqq m\leqq k+h$, and define $\lambda\triangleq m/(k+h)$, $\frac{1}{2}\leqq\lambda\leqq 1$, and

$$\delta\triangleq\frac{1-\lambda}{\lambda H(\mu)};$$

that is, $\lambda=(1+\delta H(\mu))^{-1}$ and, by (27), $r=\lfloor\delta\cdot m\rfloor$.

For any $\theta_1>0$ there exists an integer $N_1(\theta_1)$ such that

$$\begin{aligned}\log_2 V(m-2,r-1)&=\log_2 V(\lambda(k+h)-2,\lfloor\delta\cdot\lambda(k+h)\rfloor-1)\\&\geqq\lambda(k+h)\cdot(H(\delta)-\theta_1)\end{aligned}$$

whenever $\lambda(k+h)\geqq N_1(\theta_1)$. Since $\frac{1}{2}\leqq\lambda\leqq 1$, we can set τ to be at least $2^{2N_1(\theta_1)+1}$, in which case

$$V(m-2,r-1)\geqq 2^{(k+h)\cdot(\lambda H(\delta)-\theta_1)}. \tag{28}$$

Hence, whenever $t/\varepsilon \geqq \tau$ we have

$$V(m-2, r-1) \geqq \tfrac{1}{4} \cdot (t/\varepsilon)^{\beta}, \tag{29}$$

where β is any constant satisfying

$$\beta \leqq \frac{H(\delta)}{1+\delta H(\mu)} - \theta_1 = \psi(\delta, \mu) - \theta_1$$

(recall the notations of Lemma 5.5).

Plugging the values of λ, μ, and δ into (c), we obtain the following condition

$$V(\lambda(k+h), (1-\mu) \cdot \delta \cdot \lambda(k+h)) \leqq 2^h, \tag{30}$$

which implies (c). By Lemma 5.4, (30) is satisfied when $\lambda(k+h) \cdot H((1-\mu)\delta) \leqq h$, and, by the definition of ρ, we thus obtain the condition

$$\rho \geqq \lambda \cdot H((1-\mu)\delta) = \frac{H((1-\mu)\delta)}{1+\delta H(\mu)} = \omega_2(\delta, \mu), \tag{31}$$

i.e., $(\delta, \mu) \in D_2(\rho)$ (see Eq. (16)). Hence, at this point, (31), together with the definition of r in (27), guarantee conditions (b) and (c).

Refer now to condition (a). Clearly, $m \leqq n$ since we require $m \leqq k+h \leqq n$. As for the lower bound on m, it can be easily verified that the inequality $k \leqq m = \lambda(k+h)$ is satisfied if

$$\rho \geqq \frac{\delta H(\mu)}{1+\delta H(\mu)} = \omega_1(\delta, \mu),$$

i.e., $(\delta, \mu) \in D_1(\rho)$.

Now, let θ be fixed in the interval $(0, 1]$ and let ρ_0 satisfy $\gamma(\rho_0) = \theta/2$. We distinguish between the following cases.

Case 1. $0 \leqq \rho < \rho_0$. In this case we have $\alpha(t, \varepsilon) - \theta \leqq \gamma(\rho) - \theta/2 \leqq 0$ and, therefore, the theorem follows from the information bound $L(n, t, \varepsilon) = \Omega(\log n)$ (Eq. (7)), which holds for any $t \geqq 1$ and $\varepsilon \leqq \frac{1}{2}$.

Case 2. $\rho_0 \leqq \rho \leqq 1-\theta$. Let $(\delta = \delta(\rho), \mu = \mu(\rho))$ be a pair which attains the maximum in (17). Note that, since $(\delta, \mu) \in D_1(\rho) \cap D_2(\rho)$, conditions (a), (b), and (c) are satisfied. By Remark 5.2 we also have

$$\rho = \omega_2(\delta, \mu) \leqq \psi(\delta, \mu) < \psi(\delta, \mu) + \omega_1(\delta, \mu)$$

for all $\rho_0 \leqq \rho \leqq 1-\theta$. Therefore, there exists a positive constant θ_2, depending only on θ, such that $\rho \leqq \psi(\delta, \mu) + \omega_1(\delta, \mu) - \theta_2$. This allows us to bound $\sigma(m, k, r)$ from above by

$$\begin{aligned}
\log_2 \sigma(m, k, r) &= m - k - 1 - \log_2 V(m-2, r-1) \\
&\overset{(28)}{\leqq} \frac{k+h}{1+\delta H(\mu)} - k - 1 - (k+h)(\psi(\delta, \mu) - \theta_1) \\
&\leqq (k+h)(\rho - \psi(\delta, \mu) - \omega_1(\delta, \mu) + \theta_1) - 1 \\
&\leqq -(k+h)(\theta_2 - \theta_1) - 1 \leqq -k \cdot \frac{\theta_2 - \theta_1}{1-\rho_0} - 1
\end{aligned}$$

(assuming $\theta_2 \geqq \theta_1$). Hence, we can set $\theta_1 = \frac{1}{2}\min(\theta, \theta_2)$, $\tau \geqq 2(1-\rho_0)/\theta_2$, and, by Lemma 5.5, $\beta = \gamma(\rho) - \theta/2 \leqq \gamma^*(\rho) - \theta_1$ (in (29)), yielding

$$V(m-2, r-1) \geqq \tfrac{1}{4} \cdot (t/\varepsilon)^{\gamma(\rho)-\theta/2} \geqq \tfrac{1}{4} \cdot (t/\varepsilon)^{\alpha(t/\varepsilon)-\theta}$$

and (assuming $\tau \geqq 1$)

$$\sigma(m, k, r) \leqq \sqrt[\tau]{\varepsilon}.$$

Case 3. $1-\theta<\rho\leqq 1$. Set $r=m=h$ and $l=0$; for these values we have $V(m-2, r-1)=\frac{1}{4}\cdot 2^h$, and it is easy to check that conditions (b) and (c) are satisfied. Condition (a) holds since, for this range of ρ, we have $k\leqq h<n$ (assuming $\theta\leqq\frac{1}{2}$). On the other hand,

$$(t/\varepsilon)^{\alpha(t,\varepsilon)-\theta}\leqq(t/\varepsilon)^{1-\theta}\leqq 4\cdot 2^{(k+h)(1-\theta)}=4\cdot 2^{(h/\rho)(1-\theta)}\leqq 4\cdot 2^h$$

and

$$\sigma(m,k,r)=\frac{2^{m-k-1}}{2^{m-2}}=2^{1-k}<4\varepsilon.$$

In Cases 2 and 3 we thus have

$$V(m-2,r-1)=\Omega((t/\varepsilon)^{\alpha(t,\varepsilon)-\theta})$$

and

$$\sigma(m,k,r)\leqq\sqrt[\tau]{4\varepsilon}.$$

Recalling that $\log(n-m+2)\geqq\log(n+2-k-h)\geqq\log(n+2-\log_2(t/\varepsilon))$, the theorem is now implied by (26). □

Remark 5.4. We now consider briefly the case where $t/\varepsilon>2^n$, $\varepsilon\geqq 2^{-n}$. Referring to the notations of the last proof, our proof fails if the optimal value for m turns out to be greater than n. When $k+h\leqq n(1+\theta/2)$ we can still repeat the proof with $k'=\lfloor k/(1+\theta/2)\rfloor$ and $h'=\lfloor h/(1+\theta/2)\rfloor$, yielding

$$L(n,t,\varepsilon)=\Omega((t/\varepsilon)^{\alpha(t,\varepsilon)-\theta}).$$

Assume now that $k+h>n(1+\theta/2)$. Recalling that $m=(k+h)/(1+\delta H(\mu))$, we thus have to add the following condition

$$1+\delta H(\mu)\geqq\frac{k+h}{n} \tag{32}$$

to conditions (a)-(c) while maximizing $V(m-2,r-1)$ in (26) (note that, since $k\leqq n$, equality in (32) implies condition (a)). Instead of going through the steps of the proof of Theorem 5.1, we can obtain a simpler bound by assuming equality in both (31) and (32), resulting in the combined condition

$$\frac{1+\delta H(\mu)}{k+h}=\frac{1}{n}=\frac{H((1-\mu)\delta)}{h}. \tag{33}$$

For $h<n$ there exists a unique solution (δ,μ) to (33), satisfying

$$\delta=\frac{H^{-1}(\frac{h}{n})}{1-\mu}=\frac{\frac{k}{n}+\frac{h}{n}-1}{H(\mu)},$$

and when $h=n$ we can take $\delta=1$. The lower bound is now obtained by plugging the solution for δ into the right-hand side of

$$L(n,t,\varepsilon)\geqq V(m,r)-2^{m-k}+1=V(n,\lfloor\delta\cdot n\rfloor)-2^{n-k}+1,$$

the latter bound being a simplified version of Lemma 5.3. Finally, noting that $n-k<h-n\theta/2=n(H((1-\mu)\delta)-\theta/2)$, we have

$$L(n,t,\varepsilon)\geqq 2^{n(H(\delta)-\theta/4)}-2^{n(H((1-\mu)\delta)-\theta/2)}=\Omega(2^{n(H(\delta)-\theta)})$$

for every fixed θ and for sufficiently large n (compare with part (ii) of Theorem 5.2).

Proof of Theorem 5.2. (i) The proof is similar to that of Lemma 5.3. Let $L \triangleq L_{\mathrm{DNF}}(n, t, \varepsilon)$ and let B be an $L \times n$ binary matrix whose rows are the elements of a DNF ε-approximation set of size L. Let $k \triangleq \lfloor -\log_2 \varepsilon \rfloor$ ($\geqq 2$), $h \triangleq \lfloor \log_2 t \rfloor$, and $m \triangleq k+h+1$, and let C be an $L \times m$ submatrix of B consisting (say) of the last m columns of B. We show that C contains among its rows at least $2^m - 2^{h+1}+1$ distinct vectors of $\{0, 1\}^m$.

Assume, to the contrary, that 2^{h+1} distinct row m-vectors are missing from C, and let $\phi_i = \prod_{j=0}^{m-1} y_{i,j}$, $1 \leqq i \leqq 2^{h+1}$, $y_{i,j} \in \{x_j, \bar{x}_j\}$, be as defined in the proof of Lemma 5.3. Let $\vee$ denote the inclusive OR operation and define the two functions

$$f = \bigvee_{i=1}^{2^h} \phi_i \quad \text{and} \quad g = \bigvee_{i=2^h+1}^{2^{h+1}} \phi_i,$$

both over $\{0, 1\}^n$. Note that for $i \neq l$, ϕ_i and ϕ_l (regarded as functions over $\{0, 1\}^n$) do not take the value 1 simultaneously at any point of $\{0, 1\}^n$. Therefore, the truth table of $\phi = f+g$ is of weight $2^{h+1} \cdot 2^{n-m} = 2^{-k} \cdot 2^n \geqq \varepsilon \cdot 2^n$. On the other hand, our contrary assumption implies that ϕ takes the zero value at every evaluation point. It follows that the truth tables of f and g coincide at each evaluation point, in spite of the fact that they are ε-far.

Substituting $r = m = k+h+1 \leqq \log_2 (t/\varepsilon)+1 < n$ and $s = 2^{h+1}-1$ in Lemma 5.2, we obtain,

$$\begin{aligned} L &\geqq 2^{k+h-1} \cdot \left(\log_2 (n+1-k-h) - \log_2 \left(1 + \frac{(2^{h+1}-1)(n-k-h)}{2^{k+h}} \right) \right) \\ &\geqq 2^{k+h-1} \cdot \log_2 \left(\frac{1}{n+1-k-h} + \frac{1}{2^{k-1}} \right)^{-1} \qquad (34) \\ &> \frac{1}{8} \cdot (t/\varepsilon) \cdot \log_2 \left(\frac{1}{n+1-\log_2 (t/\varepsilon)} + 4\varepsilon \right)^{-1}. \end{aligned}$$

(ii) Suppose that $\varepsilon < \frac{1}{2}$ and that $t/\varepsilon \geqq 2^{n-1}$. The idea is to find t' and ε' such that $t' \leqq t$, $\varepsilon \leqq \varepsilon' < \frac{1}{2}$, and $2^{n-2} \leqq t'/\varepsilon' < 2^{n-1}$. Having done that, we substitute $k' \triangleq \lfloor -\log_2 \varepsilon' \rfloor$ and $h' \triangleq \lfloor \log_2 t' \rfloor$ in (34), thus yielding

$$\begin{aligned} L_{\mathrm{DNF}}(n, t, \varepsilon) &\geqq L_{\mathrm{DNF}}(n, t', \varepsilon') \\ &\geqq 2^{k'+h'-1} \cdot \log_2 \left(\frac{1}{n+1-k'-h'} + \frac{1}{2^{k'-1}} \right)^{-1} \\ &\geqq 2^{n-4} \cdot \log_2 \left(\frac{1}{n+1-k'-h'} + \frac{1}{2^{k'-1}} \right)^{-1} \\ &\geqq \left(\frac{1}{16} \cdot \log_2 \frac{6}{5} \right) \cdot 2^n. \end{aligned}$$

Indeed, assuming that $n \geqq 4$, set $\varepsilon' = \max(\varepsilon, 2^{2-n})$ $(< \frac{1}{2})$ and $t' = \lceil \varepsilon' \cdot 2^{n-2} \rceil$. We thus have $t'/\varepsilon' \geqq 2^{n-2}$, $\varepsilon' \geqq \varepsilon$ and $t' = \max(1, \lceil \varepsilon \cdot 2^{n-2} \rceil) \leqq \max(1, \varepsilon \cdot 2^{n-1}) \leqq t$ (assuming $t \neq 0$). Furthermore,

$$\frac{t'}{\varepsilon'} < \frac{\varepsilon' \cdot 2^{n-2}+1}{\varepsilon'} = 2^{n-2} + \frac{1}{\varepsilon'} \leqq 2^{n-1},$$

as required. □

The discussion in this section can be extended easily to the adaptive scheme as well. In particular, the proof of Lemma 5.3 and, consequently, of Theorem 5.1 and

Theorem 5.2, apply also to this case, except that the approximated function is now $2t$-sparse (or $2t$-term DNF).

6. Probabilistic polynomial-time approximation algorithm. In this section we describe an algorithm for finding ε-approximations of t-sparse functions over $\{0, 1\}^n$. The algorithm is adaptive and, given a (prespecified) probability p of failure, its running time is $O((t^2n^2/\varepsilon)\cdot\log(tn/p))$ bit operations. The approximation is carried out by actually finding monomials of the underlying function which are "short," i.e., each is a product consisting of few variables.

Given a function $f:\{0,1\}^n\to\{0,1\}$, for every $\mathbf{u}\in\{0,1\}^l$, $0\leqq l\leqq n$, we associate a function $f_{\mathbf{u}}$ defined as follows: For $l=0$ we have $f_{[\,]}\triangleq f$, where [ ] denotes the "empty vector." Now, given $f_{\mathbf{u}}$, $\mathbf{u}\in\{0,1\}^l$, $l<n$, we define $f_{[\mathbf{u}0]}$ and $f_{[\mathbf{u}1]}$ by the unique decomposition

$$f_{\mathbf{u}}(x_{n-l-1}, x_{n-l-2}, \ldots, x_0) = f_{[\mathbf{u}0]}(x_{n-l-2}, \ldots, x_0) + x_{n-l-1}\cdot f_{[\mathbf{u}1]}(x_{n-l-2}, \ldots, x_0) \tag{35}$$

(see Eq. (2), § 2). In particular, when $l=n$, $f_{\mathbf{u}}$ is a coefficient of f. We now define a binary directed tree $T(f)$ whose vertices correspond to $f_{\mathbf{u}}$, $\mathbf{u}\in\bigcup_{l=0}^{n}\{0,1\}^l$, and for $l\neq n$ we have edges directed from $f_{\mathbf{u}}$ to the two vertices $f_{[\mathbf{u}0]}$ and $f_{[\mathbf{u}1]}$. Clearly, $f=f_{[\,]}$ is the root of $T(f)$ and $f_{\mathbf{u}}$, $\mathbf{u}\in\{0,1\}^n$, are its leaves. Now, let W be a binary subtree of $T(f)$ growing from the root $f_{[\,]}$ and let $\Lambda(W)$ denote the set of leaves of W. Using the notation $\mathbf{x}^{\mathbf{u}}$, $\mathbf{u}=[u_{l-1}u_{l-2}\ldots u_0]\in\{0,1\}^l$, for $x_{n-1}^{u_{l-1}}x_{n-2}^{u_{l-2}}\cdots x_{n-l}^{u_0}$, we can write, by (35), the identity

$$f = \sum_{\mathbf{u}\ \text{s.t.}\ f_{\mathbf{u}}\in\Lambda(W)} \mathbf{x}^{\mathbf{u}}\cdot f_{\mathbf{u}} \tag{36}$$

for any binary subtree W of $T(f)$. Note that if f is t-sparse, $\Lambda(W)$ contains at most t leaves which are not identically zero.

The heart of our algorithm (procedure APPROX in Fig. 3) is a partial *Depth First Search* (DFS) on the vertices of $T(f)$, starting at the root $f_{[\,]}$ and resulting in a binary subtree W_{DFS} of $T(f)$. Using randomization, we "weigh" the truth table of $\mathbf{x}^{\mathbf{u}}\cdot f_{\mathbf{u}}$; if its *relative weight* (= the weight of a truth table divided by its size) turns out to be less than $\theta\triangleq\varepsilon/t$, we climb back to the father of $f_{\mathbf{u}}$ and, in this case, $f_{\mathbf{u}}$ becomes a so-called *negligible* leaf of W_{DFS}. Otherwise, we continue down into $T(f)$ unless $f_{\mathbf{u}}$ is a leaf of $T(f)$ (and, thus, of W_{DFS}); in this case $f_{\mathbf{u}}\equiv 1$ and, so, we have found a monomial $\mathbf{x}^{\mathbf{u}}$ of f. Such a leaf $f_{\mathbf{u}}$, referred to as a *terminal* leaf, is now added to the approximating function of f. Exploring the subtree growing down from a nonnegligible vertex $f_{\mathbf{u}}$ in $T(f)$, we then climb back to the father of $f_{\mathbf{u}}$.

The above procedure is implemented in APPROX as follows. The input parameters are n, t, ε, and the allowed probability p of failure. We also assume that there exists a subroutine ("oracle") which, given $\mathbf{z}\in\{0,1\}^n$, returns the value of the underlying function f at $\mathbf{z}$. The output of APPROX is an ε-approximation $\hat{f}$ of f, represented by its nonzero monomials. The main module in APPROX consists of one call to the procedure DFS which traverses $T(f)$, inducing the subtree W_{DFS}.

The routine DFS is recursive, and at each recursion level we regard the vertex $f_{\mathbf{u}}$, $\mathbf{u}\in\{0,1\}^l$ (the input parameter to DFS) as a root of the subtree growing down from $f_{\mathbf{u}}$. The truth table of $f_{\mathbf{u}}$ is weighed by a procedure named NONZERO, which checks whether the relative weight of $\mathbf{x}^{\mathbf{u}}\cdot f_{\mathbf{u}}$ is at least ε/t, using not-too-many samples of $f_{\mathbf{u}}$. The specifications of NONZERO are as follows:

(a) If the relative weight of $\mathbf{x}^{\mathbf{u}}\cdot f_{\mathbf{u}}$ is at least $\theta=\varepsilon/t$, NONZERO returns "true" with probability $\geqq 1-q$, where $q\triangleq p/(tn+1)$.

```
procedure APPROX(f; n, t, ε, p) output: f̂;
/*
    ε-approximation of a t-sparse Boolean function f over {0, 1}^n.
    p is an upper bound on the probability of failure.
    f̂ is the approximating function of f.
    We assume the existence of a subroutine ("oracle") which, given z ∈ {0, 1}^n, returns f(z).
*/
begin
    f̂ ← 0; θ ← ε/t; q ← p/(tn+1);
    /*  n, t, θ, q, and f̂ are global for all subsequent routines.  */
    DFS([], 0)
end;
procedure DFS(u, l);
/*  Perform DFS starting at the vertex f_u, u ∈ {0, 1}^l.  */
begin
    if NONZERO(u, l) then
        if l = n then  /*  a terminal leaf  */
            f̂ ← f̂ + x^u
        else begin DFS([u 0], l+1); DFS([u 1], l+1) end
end;
procedure NONZERO(u, l) output: "true" / "false";
/*  Check whether the relative weight of x^u · f_u is at least θ.  */
begin
    if 2^{-w(u)} < θ then
        NONZERO ← "false"
    else begin
        i ← 0;
        N ← ⌈(log q)/(log (1 − θ · 2^{w(u)}))⌉;  /*  2^{-w(u)} = θ ⇒ N = 0.  */
        repeat
            i → i+1;
            choose at random z ∈ {0, 1}^{n−l};
            b ← Σ_{v s.t. v⊑u} f([v z])
        until (i ≧ N) or (b = 1);
        NONZERO ← (b = 1)
    end
end;
```

FIG. 3. *ε-approximation of Boolean functions.*

(b) If $f_{\mathbf{u}}$ is identically zero, NONZERO always returns "false."
(Note that the output of NONZERO is unspecified if the relative weight of a nonzero $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is less than θ.) The value assigned to q guarantees an overall probability of failure which is not greater than p. In case NONZERO returns a "false" answer on weighing $f_{\mathbf{u}}$ (meaning that the relative weight of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is, most likely, smaller than ε/t), we return to the father of $f_{\mathbf{u}}$. The same holds also when NONZERO returns "true" and $f_{\mathbf{u}}$ is a terminal leaf, in which case the monomial $\mathbf{x}^{\mathbf{u}}$ is added to the approximating function $\hat{f}$. Otherwise, we continue down the tree $T(f)$.

We now consider the implementation of the routine NONZERO, in view of the above specifications (a) and (b). Given $\theta = \varepsilon/t$, $q = p/(tn+1)$, and the parameter $\mathbf{u} \in \{0,1\}^l$, we pick at random a set P of $N = \lceil (\log q)/\log(1-\theta \cdot 2^{w(\mathbf{u})}) \rceil$ points in $\{0,1\}^{n-l}$ and then ask for the values of f at the set $Q_{\mathbf{u}} \subseteq \{0,1\}^n$ defined by

$$Q_{\mathbf{u}} = \{[\mathbf{v}\,\mathbf{z}] \mid \mathbf{v} \in \{0,1\}^l, \mathbf{v} \sqsubseteq \mathbf{u}, \text{and } \mathbf{z} \in P\}. \tag{37}$$

By Remark 2.1, it is easy to verify that the values of $f_{\mathbf{u}}$ at P are given by

$$f_{\mathbf{u}}(\mathbf{z}) = \sum_{\mathbf{v} \text{ s.t. } \mathbf{v} \sqsubseteq \mathbf{u}} f([\mathbf{v}\,\mathbf{z}]), \qquad \mathbf{z} \in P.$$

Sampling the truth table of $f_{\mathbf{u}}$ in this manner, NONZERO returns "false" if and only if $f_{\mathbf{u}}$ vanishes at P. The choice of N guarantees an error probability $\leqq q$ for answering "false" instead of "true." Note that when $2^{-w(\mathbf{u})} < \theta$, the relative weight of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is definitely smaller than θ and, therefore, no sampling is required. In the special case when $2^{-w(\mathbf{u})} = \theta$ (in which case $N=0$) we take one sampled value of $f_{\mathbf{u}}$. If this value is zero, the relative weight of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is proven to be less than θ and, therefore, $f_{\mathbf{u}}$ becomes a negligible leaf. Otherwise, NONZERO returns "true."

We now state the validity and the time complexity of APPROX.

LEMMA 6.1. *The procedure* NONZERO *complies with the above specifications* (a) *and* (b).

Proof. First, note that NONZERO returns "true" only when it actually samples a nonzero value of $f_{\mathbf{u}}$. Therefore, when $f_{\mathbf{u}}$ is identically zero, NONZERO will always return "false," thus establishing requirement (b). As for requirement (a), assume that the relative weight of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is at least θ. This means that the relative weight of $f_{\mathbf{u}}$ is at least $\theta \cdot 2^{w(\mathbf{u})}$ and, therefore, the probability of having N zero samples of $f_{\mathbf{u}}$ is not greater than $(1-\theta \cdot 2^{w(\mathbf{u})})^N$, which, due to the choice of N, is not greater than q (this applies also to the special case when $\theta = 2^{-w(\mathbf{u})}$, where the truth table of $f_{\mathbf{u}}$ is all-one and, therefore, NONZERO will return "true" due to the one sample it makes). □

LEMMA 6.2. (i) DFS *traverses at most* $2t(n-1)+3$ *vertices of* $T(f)$; (ii) *at most* $tn+1$ *of these vertices correspond to nonzero functions* $f_{\mathbf{u}}$.

Proof. Every nonzero vertex $f_{\mathbf{u}}$ in $T(f)$ is situated on a path from the root $f_{[]}$ to some nonzero leaf of $T(f)$. Since the number of such nonzero leaves is at most t, the number of nonzero vertices in $T(f)$ is at most $tn+1$, which is also an upper bound on the number of nonzero vertices in any subtree of $T(f)$. This proves part (ii) of the lemma.

As for part (i), let $f_{\mathbf{u}}$ be an inner vertex in W_{DFS}, i.e., a vertex which is not a leaf. Clearly, $f_{\mathbf{u}} \neq 0$, or else, by Lemma 6.1, NONZERO must return "false" on weighing $f_{\mathbf{u}}$, making $f_{\mathbf{u}}$ a negligible leaf, rather than an inner leaf, of W_{DFS}. Therefore, the inner vertices of W_{DFS} are all nonzero inner vertices of $T(f)$, the number of which is at most $r = t(n-1)+1$. Hence, the total number of vertices in W_{DFS} is at most $2r+1 = 2t(n-1)+3$. □

LEMMA 6.3. *The procedure* APPROX *returns, with probability* $\geqq 1-p$, *an* ε*-approximation* $\hat{f}$ *for any n-variable t-sparse function f.*

Proof. First, note that NONZERO might return a wrong answer ("false" instead of "true") only at a vertex $f_{\mathbf{u}}$ whose corresponding function $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is of relative weight $\geqq \theta = \varepsilon/t$. By Lemma 6.2, the number of such vertices is at most $tn+1$ and, therefore, the probability of the answers of NONZERO being all correct is at least $(1-q)^{tn+1} \geqq 1-(tn+1)q = 1-p$.

Consider now an execution of APPROX where all the answers of NONZERO are correct. Let $\hat{\Lambda}$ denote the set of terminal leaves in W_{DFS}. The approximating function, computed by APPROX, is given by

$$\hat{f} = \sum_{\mathbf{u} \text{ s.t. } f_{\mathbf{u}} \in \hat{\Lambda}} \mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}.$$

Now, let $\tilde{\Lambda} \triangleq \Lambda(W_{\mathrm{DFS}}) - \hat{\Lambda}$ denote the set of all negligible leaves of W_{DFS} and define the function $\tilde{f}$ by

$$\tilde{f} = \sum_{\mathbf{u} \text{ s.t. } f_{\mathbf{u}} \in \tilde{\Lambda}} \mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}. \tag{38}$$

By (36) we have $f = \hat{f} + \tilde{f}$. It suffices to show that the relative weight of $\tilde{f}$ if less than ε. Let $f_{\mathbf{u}} \in \tilde{\Lambda}$ be a negligible leaf encountered during any of the recursion levels of DFS.

If $f_{\mathbf{u}}$ is identically zero, then the contribution of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ to $\tilde{f}$ is zero. On the other hand, if $f_{\mathbf{u}} \neq 0$, then, assuming the answers of NONZERO being all correct, the relative weight of $\mathbf{x}^{\mathbf{u}} \cdot f_{\mathbf{u}}$ is less than ε/t. Now, the number of nonzero leaves in W_{DFS} is bounded from above by the number of nonzero leaves in $T(f)$ which, in turn, is upper-bounded by t. In particular, the number of nonzero negligible leaves in W_{DFS} is at most t and, therefore, the relative weight of $\tilde{f}$ is less than $t \cdot (\varepsilon/t) = \varepsilon$. □

THEOREM 6.1. *The ε-approximation of any n-variable t-sparse function f can be performed, with probability $\leqq p$ of failure, by querying $O((t^2 n/\varepsilon) \cdot \log(tn/p))$ values of f, involving $O(n)$ bit operations per query.*

Proof. The number of queries issued at each call to NONZERO is given by (see Eq. (37))

$$(39) \qquad |Q_{\mathbf{u}}| = 2^{w(\mathbf{u})} \cdot N = 2^{w(\mathbf{u})} \cdot \lceil (\log q)/\log(1 - \theta \cdot 2^{w(\mathbf{u})}) \rceil = O((1/\theta) \cdot \log(1/q)).$$

Now, by Lemma 6.2, the number of calls to NONZERO is at most $2t(n-1)+3$. Substituting $\theta = \varepsilon/t$ and $q = p/(tn+1)$ in (39) yields a total number of $O((t^2 n/\varepsilon) \cdot \log(tn/p))$ queries. It is easy to verify that each such query in NONZERO involves $O(n)$ bit operations. □

Finally, we show how APPROX can be used to approximate any t-sparse function, without knowing the value of t in advance. In such a scheme, we perform $M \leqq \lceil \log_2 t \rceil$ iterations of APPROX; the mth iteration ($m = 1, 2, \ldots, M$) is executed with $t_m = 2^m$, and the global variables of APPROX are set to $\theta_m \leftarrow \varepsilon/(t_m(n-1)+2)$ and $q_m \leftarrow p/(t_m n + 1)$. Let $W_{\mathrm{DFS}}(m)$ denote the tree traversed by DFS during the mth iteration, and let $W^*_{\mathrm{DFS}}(m)$ denote the subtree of $W_{\mathrm{DFS}}(m)$ obtained by deleting its negligible leaves. Note that each leaf of $W^*_{\mathrm{DFS}}(m)$ corresponds to some nonzero function $f_{\mathbf{u}}$ and, therefore, the number of such leaves cannot exceed t. Now, the iteration process terminates when, for the first time, t_m becomes greater than or equal to the number of leaves of $W^*_{\mathrm{DFS}}(m)$ (in which case $m = M \leqq \lceil \log_2 t \rceil$). By arguments similar to those given in the proof of Lemma 6.2, the number of inner vertices in $W_{\mathrm{DFS}}(M)$ is not greater than $t_M(n-1)+1$ and, therefore, the number of negligible leaves in $W_{\mathrm{DFS}}(M)$ is at most $t_M(n-1)+2$. It thus follows that the relative weight of $\tilde{f}_M$, defined by (38) for the Mth iteration, is less than ε. Hence, the output $\hat{f}_M$ of the last iteration of APPROX is, with probability $\leqq p$ of failure, an ε-approximation of f. It can be easily verified that the above process involves a total of $O((t^2 n^2/\varepsilon) \cdot \log(tn/p))$ queries.

Finding the t-term DNF counterpart of the above procedure still remains an open problem. The approximation problem in the t-term DNF case has also been dealt with in the literature in a wider context, namely, when the approximation factor ε is measured according to an arbitrary distribution induced on $\{0, 1\}^n$ (and not necessarily according to the uniform distribution). For related work see, for instance, [8], [11], [13]-[15].

Acknowledgment. The authors thank Noga Alon for the helpful discussions and the anonymous referees for their valuable suggestions which helped improve the presentation of this paper. In particular, the approximation procedure for the case where t is unknown, discussed in § 6, is due to one of the referees.

REFERENCES

[1] D. ANGLUIN, *Learning k-term* DNF *formulas using queries and counterexamples*, Res. Report RR-559, Department of Computer Science, Yale University, New Haven, CT, 1987.

[2] D. ANGLUIN AND C. H. SMITH, *Inductive inference: Theory and methods*, Comput. Surveys, 15 (1983), pp. 237-269.

[3] M. BEN-OR AND P. TIWARI, *A deterministic algorithm for sparse multivariate polynomial interpolation*, in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 301–309.

[4] M. CLAUSEN, A. DRESS, J. GRABMEIER, AND M. KARPINSKI, *On zero-testing and interpolation of k-sparse multivariate polynomials over finite fields*, Report 8522-CS, Institut für Informatik, Universität Bonn, Bonn, FRG, 1988.

[5] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[6] D. Y. GRIGORIEV, M. KARPINSKI, AND M. F. SINGER, *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, Report 8523-CS, Institut für Informatik, Universität Bonn, Bonn, FRG, 1988.

[7] L. HELLERSTEIN AND M. WARMUTH, private communication.

[8] M. KEARNS, M. LI, L. PITT, AND L. G. VALIANT, *On the learnability of Boolean formulae*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 285–295.

[9] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.

[10] R. J. MCELIECE, *The Theory of Information and Coding*, Addison-Wesley, Reading, MA, 1977.

[11] L. PITT AND L. G. VALIANT, *Computational limitations on learning from examples*, Tech. Report TR-05-86, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1986.

[12] G. SEROUSSI AND N. H. BSHOUTY, *Vector sets for exhaustive testing of logic circuits*, IEEE Trans. Inform. Theory, IT-34 (1988), pp. 513–522.

[13] L. G. VALIANT, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[14] ———, *Learning disjunctions of conjunctions*, in Proc. 9th Internat. Joint Conference on Artificial Intelligence, 1 (1985), pp. 560–566.

[15] ———, *Deductive learning*, Aiken Computational Laboratory, Harvard University, Cambridge, MA, 1984.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 315-327, April 1991

© 1991 Society for Industrial and Applied Mathematics
009

LOWER BOUNDS FOR COMPUTATIONS WITH THE FLOOR OPERATION*

YISHAY MANSOUR†, BARUCH SCHIEBER‡, AND PRASOON TIWARI‡§

Abstract. A general lower bound technique is developed for computation trees with operations $\{+, -, *, /, \lfloor\cdot\rfloor, <\}$ and constants $\{0, 1\}$, for functions that have as their input a single n-bit integer. The technique applies to many natural functions, such as *perfect square root* (deciding if the square root of the input is integral or not), computing the parity of $\lfloor\log x\rfloor$, etc. The arguments are then extended to obtain the same lower bounds on the time complexity of any RAM program with operations $\{+, -, *, /, \lfloor\cdot\rfloor, <\}$ that solves the problem. Another related result is described in a companion paper [*Proc. 29th IEEE Symposium on Foundations of Computer Science*, 1988] and [*J. Assoc. Comput. Mach.*, 1991, to appear].

Key words. algorithms, lower bound, square-root, Newton iteration, mod operation, floor operation, truncation

AMS(MOS) subject classifications. 68Q05, 68Q25, 68Q40

1. Introduction. Much effort has been devoted in recent years towards proving lower bounds on the running time of algorithms that involve arithmetical operations and comparisons. (See, e.g., [Sto76], [PS81], [Sch82], [SY82], [Ben83], [DO85].) However, several difficulties are encountered when we try to extend these lower bounds to more realistic computation models. First, all of the above-mentioned lower bounds assumed that the set of available arithmetic operations is $\{+, -, *, /\}$,[1] and the proof techniques break down when the **floor** operation (or the equivalent *integer* division operation) is in our repertoire. Including the **floor** operation in the repertoire increases the power of the model considerably. For example, Stockmeyer [Sto76] proves that when the set of available operations is $\{+, -, *, /\}$, deciding if an n-bit integer is odd or even takes $\Theta(n)$ time. This can be done in *constant* time when the **floor** operation is in our repertoire. Second, most of the known lower bounds use the computation (or the decision) tree model. This computation model does not have the capability of *indirect addressing*, while most of the realistic models do have this capability. Finally, all the above-mentioned techniques are applicable only when the inputs are either real or rational numbers, and cannot be applied to problems where the inputs are restricted to be n-bit integers, where n is the size of the problem at hand. In this paper we attempt to tackle some of these difficulties.

The models of computation considered in this paper are the computation tree model [Str72], [Str83], [Ben83], and the Random Access Machine (RAM) model [Sch79], [AHU74]. A detailed description of each of these models is given in the next section. It appears that the two computation models are incomparable. On one hand,

* Received by the editors December 27, 1988; accepted for publication (in revised form) July 9, 1990.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The work of this author was supported by an International Sephardic Education Foundation fellowship and National Science Foundation contract 8657527-CCR. Part of the research of this author was done while visiting IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

‡ IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

§ Present address, Department of Computer Science, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, Wisconsin 53706.

[1] Throughout the paper "/" stands for *exact* division, i.e., $8/5 = 1.6$. All sets of operations in this paper are assumed to include the "<" comparison operation, which is never explicitly specified.

a computation tree is a nonuniform model of computation, while a RAM is the corresponding uniform model of computation, and hence, weaker in this sense. On the other hand, a RAM is capable of indirect addressing while a computation tree is not. So far, the power of indirect addressing has not been fully characterized, and it is not known whether it is a substantial theoretical advantage.

The proof techniques of this paper apply to a large class of functions. This is the class of $M(n)$*-invariant* functions defined as follows: A boolean function $f(\cdot)$ is said to be $M(n)$-invariant if for any $\lambda < M(n)$, there exist two positive n-bit integers a_0 and a_1, both divisible by λ, such that $f(a_0) \neq f(a_1)$. Many functions are $M(n)$-invariant for a suitable choice of $M(n)$. For example, it is easy to verify that the following *Perfect Square Function* is $2^{n/2-1}$-invariant: given an integer a, output "1" if the square root of a is an integer (in this case we say that a is a *perfect square*) and "0" otherwise. Another example is the function $(\lfloor \log x \rfloor \bmod 2)$, which is 2^{n-1}-invariant.

We prove an $\Omega(\sqrt{\log \log M(n)})$ lower bound[2] on the depth of any computation tree with operations $\{+, -, *, /, \lfloor \cdot \rfloor\}$ and constants $\{0, 1\}$,[3] that computes an $M(n)$-invariant function for all n-bit integer inputs. If $M(n)$ is not a constant function, then this implies that there is no finite depth computation tree with operations $\{+, -, *, /, \lfloor \cdot \rfloor\}$ and the rational numbers as the set of allowable constants that computes an $M(n)$-invariant function for all integers. We then extend the arguments to obtain the same lower bounds on the time complexity of any RAM program with operations $\{+, -, *, /, \lfloor \cdot \rfloor\}$. It should be noted that the assumption that only the constants $\{0, 1\}$ are available is essential to the lower bound proof. In contrast, in a companion paper [MST89], we show that, on n-bit integer inputs, every function can be evaluated in $O(1)$ steps if arbitrary constants (that depend on n) are available. The reason is that every function of one variable, when restricted to n-bit integer inputs, can be represented as a polynomial of degree 2^n; and these polynomials can be evaluated using $O(1)$ operations from the set $\{+, -, *, /, \lfloor \cdot \rfloor\}$ if arbitrary constants are available.

While working on their book [GLS88], Grötschel, Lovász, and Schrijver asked if there exists a strongly polynomial algorithm for computing the greatest common divisor (gcd) of two integers when the **floor** operation is in the repertoire. (See [GLS88, pp. 32–33 and 225].) This problem was settled in the negative in a companion paper [MST88b]. The main result in that paper is: any computation tree with operations $\{+, -, *, /, \bmod\}$ and constants $\{0, 1\}$ that computes the gcd of two n-bit integers a and b must have depth $\Omega(\log \log n)$. The same lower bound holds also for deciding whether two given integers a and b are relatively prime, for any two n-bit integers a and b. (Observe that the **floor** and **mod** operations can implement each other in a constant number of steps.) In this paper a similar technique is applied to other problems. The major difference is that the functions considered in this paper have a single input, compared to two inputs for the gcd problem. This enables us to prove an $\Omega(\sqrt{\log n})$ lower bound, compared to the $\Omega(\log \log n)$ lower bound in [MST88b]. The gap between the lower bound in our case (e.g., when applied to the Perfect Square Function) and the known logarithmic upper bound is only quadratic, which is much smaller than the *doubly exponential* gap for the gcd problem.

The lower bound proof given here for the RAM model can be extended to the case of more than one input. Hence, by incorporating the results here with the results from [MST88b] we can obtain an $\Omega(\log \log n)$ bound on the time complexity of any RAM program with operations $\{+, -, *, /, \lfloor \cdot \rfloor\}$ and constants $\{0, 1\}$ that decides

[2] The base of all logarithms in the paper is two.

[3] Any constants, other than zero and one, must be computed explicitly.

whether two given integers a and b are relatively prime, for any two n-bit integers a and b.

For the *algebraic* tree model (i.e., trees that use only *algebraic* operations, including $+$, $-$, $*$, and $/$), good lower bound techniques are known when all the inputs are either real or rational numbers. These techniques are applied in [SY82] for algebraic decision trees and in [Ben83] for algebraic computation trees. On the other hand, few results are known when the inputs are restricted to be integers (as in our case). Among them is [Yao89], which extends the results in [Ben83] to this case. Even less is known when the set of operations contains the **floor** operation.

The power of the **floor** operation has been considered in several papers. In [BJM88] the model of an *analytic* computation tree is introduced. (An analytic computation tree operates on real numbers and has the **floor** operation in its repertoire.) Using topological arguments, that paper shows that there are certain classes of languages which cannot be recognized by analytic computation trees. The expressive power of computation trees with various sets of operations is compared in [JMW89], and the same paper also contains lower bounds for computation trees with operations $\{+, -, \mathrm{DIV}_c\}$, where DIV_c denotes integer division by constants. Their proof technique is based on methods from *Geometry of Numbers* [Cas71]. [LM85] proves an $\Omega(\log p)$ lower bound on the depth of any computation tree with operations $\{+, -, *, \lfloor\cdot\rfloor\}$ that decides whether a given triangle in the plane contains a point with integer coordinates, where p is the (binary) length of the input.

For the RAM model, Paul and Simon in [PS81] prove an $\Omega(n \log n)$ lower bound for sorting n integers on RAMs with operations $\{+, -, *, /\}$. (See also [DO85].) Their result also applies to RAMs with indirect addressing. The power of RAMs has been extensively studied in [Sch79] and [Sim81]. [IMR83] sheds more light on the nature of the **floor** operation and indirect addressing. Other relevant references are [Cob66], [Pat72], and [Kun73].

The paper is organized as follows. Section 2 includes some preliminary definitions. Section 3 is devoted to the proof of the $\Omega(\sqrt{\log\log M(n)})$ lower bound on the depth of any computation tree with operations $\{+, -, *, /, \lfloor\cdot\rfloor\}$, that computes $M(n)$-invariant functions. In § 4 we extend the lower bound to the RAM model. For completeness, we describe in § 5 the known upper bound for computing the Perfect Square Function. Finally, in § 6 we summarize our results and list some open problems.

2. Preliminaries. In this section, we first recall the definition of the computation models used in this paper. Then, we define some properties of polynomials and rational expressions. Finally, we define a lexicographic order on the set of polynomials and prove some of its properties.

2.1. The computation models. We assume that the reader is familiar with the computation tree model (see, e.g., [Str83], [Ben83]) and the RAM model (see, e.g., [AHU74], [Sch79]) used in this paper. Below, we briefly recall these models, and define some additional terminology used throughout the paper in relation to these models.

The computation tree model. A computation tree T for a one input problem is a tree with labeled vertices. The label of a vertex v is denoted f_v. The tree T has four types of vertices:

(1) *An input vertex*: The root of the tree is the input vertex and it is labeled with the single input.

(2) *Computation vertices*: Each computation vertex ν has only one child, and is labeled with either a binary operation and its two operands: $g \circ h$, or a unary operation and its operand: $\circ g$. Throughout this paper, $g, h \in \mathscr{C} \cup \{f_\mu \mid \mu$ is an ancestor of ν in $T\}$ and $\circ \in OP$, where $\mathscr{C}$ is the set of available constants, and OP is the set of available operations. In this paper, $OP = \{+, -, *, /, \lfloor \cdot \rfloor\}$, and the set $\mathscr{C}$ will be restricted to be the set $\{0, 1\}$. Note that the label f_ν of a computation vertex represents a function of the input.

(3) *Comparison vertices*: Each comparison vertex ν is labeled with $g : h$, where, again, $g, h \in \mathscr{C} \cup \{f_\mu \mid \mu$ is an ancestor of ν in $T\}$. Each comparison vertex has two children.

(4) *Output vertices*: The output vertices are the leaves of T. Each leaf ν of T is labeled with an element from the set $\{0, 1\}$.

The computation for input a starts at the root of the tree T. When it arrives at a computation vertex ν, the function f_ν is evaluated at the input a, and then the computation proceeds to the only child of ν. When the computation arrives at a comparison vertex labeled with $g : h$, the functions g and h are evaluated at the input a. The computation proceeds to the left child if $g(a) < h(a)$, and to the right child otherwise. The computation terminates at a leaf by producing the value of the label associated with it as the output.

We remark that when $OP = \{+, -, *, /\}$, we can associate a rational expression $r_\nu(x) \in \mathscr{Q}(x)$, to each computation vertex ν. This rational expression will have the following interpretation: *For all inputs a, if the computation on the tree T with the input a arrives at the vertex* ν, *then* $f_\nu(a) = r_\nu(a)$.

The Random Access Machine (RAM) model. A RAM program is a sequence $((1 : \gamma_1), (2 : \gamma_2), \cdots, (r : \gamma_r))$, where each γ_i is either (i) a *common instruction* defined in [Sch79], or (ii) an instruction which applies an operation from the set of available operations OP to a set of operands, and stores the result in a memory location. A memory location can be accessed using either *direct* addressing, that is by specifying its address explicitly, or *indirect* addressing, that is by specifying an address of a location containing its address. A RAM *without* indirect addressing is defined in a similar manner, except that no indirect addressing of the memory is allowed.

The input to a RAM program for a one input problem is given in the first memory location when the program starts. In this paper, this input is assumed to be an integer. (Note that individual bits of the input cannot be accessed in one step.)

A RAM program is said to solve a decision problem if for each instance of the problem, the first memory location contains the correct answer when the RAM executes the **halt** instruction.

Consider a RAM program $\mathscr{R}$ of time complexity h, without indirect addressing. All computations of $\mathscr{R}$ can be simulated by a computation tree $T_h^{\mathscr{R}}$ of depth $O(h)$, in a straightforward manner. *On the other hand, if* $\mathscr{R}$ *is capable of indirect addressing, no straightforward simulation by a computation tree of depth* $O(h)$ *appears possible.* In [Mey89], it is established that a RAM program $\mathscr{R}$ of time complexity h with indirect addressing can be simulated by a computation tree of depth $O(h \log h)$.

2.2. Polynomials and rational expressions. In the sequel, we consider the degree and the height of univariate polynomials and rational expressions.

The *degree* of a polynomial $P(x)$, denoted $\deg(P)$, is the maximum exponent of x appearing in any monomial of $P(x)$.

The *height* of P, denoted $\operatorname{hgt}(P)$, is the maximum among the absolute values of the coefficients of P.

For a set Λ of polynomials, the *degree* of Λ, denoted $\deg(\Lambda)$, is defined by $\max_{P\in\Lambda} \deg(P)$. Similarly, the *height* of Λ, denoted $\text{hgt}(\Lambda)$, is $\max_{P\in\Lambda} \text{hgt}(P)$.

For a rational expression $R(x)=P(x)/Q(x)$, where P and Q are polynomials, define the *degree* of R to be the larger of the degrees of P and Q. Similarly, define the *height* of R to be the larger of the heights of P and Q. Notice that cancellations are not allowed in our definition of the degree and the height of rational expressions. For example, the degree of $R(x)=P(x)/P(x)$ will be the degree of P and not zero, and the height of R will be the height of P and not one.

Let $P(x)$ and $Q(x)$ be two polynomials. Then, $P(x)\circ Q(x)$ has the usual meaning for $\circ\in\{+,-,*\}$. For $\circ=/$, $P(x)/Q(x)$ is the rational expression $R(x)=P(x)/Q(x)$. For two rational expressions $P_1(x)/Q_1(x)$ and $P_2(x)/Q_2(x)$, we define $(P_1(x)/Q_1(x))\circ(P_2(x)/Q_2(x))$ to be the rational expression

$$\frac{P_1(x)Q_2(x)\circ P_2(x)Q_1(x)}{Q_1(x)Q_2(x)}\,[4]$$

if $\circ\in\{+,-\}$; the rational expression $P_1(x)P_2(x)/Q_1(x)Q_2(x)$ if $\circ=*$; and the rational expression $P_1(x)Q_2(x)/Q_1(x)P_2(x)$ if $\circ=/$.

LEMMA 2.1. *Let $P(x)$ and $Q(x)$ be two polynomials. Then, for*

$$\circ\in\{+,-\},\ \deg(P\circ Q)\leqq\max\{\deg(P),\deg(Q)\}$$

and

$$\text{hgt}(P\circ Q)\leqq\text{hgt}(P)+\text{hgt}(Q);\ \deg(P*Q)\leqq\deg(P)+\deg(Q)$$

and

$$\text{hgt}(P*Q)\leqq\min\{(1+\deg(P)),\ (1+\deg(Q))\}\ \text{hgt}(P)\ \text{hgt}(Q).$$

Proof. We only prove the bound on $\text{hgt}(P*Q)$. Observe that when we multiply two polynomials, each of the coefficients of the product is the sum of at most $\min\{(1+\deg(P)),(1+\deg(Q))\}$ terms, each of them being the product of a coefficient in P by a coefficient in Q. The bound stated in the Lemma is an immediate consequence. □

LEMMA 2.2. *Let $R(x)=P_1(x)/P_2(x)$ and $S(x)=Q_1(x)/Q_2(x)$ be two rational expressions. Then, for $\circ\in\{+,-\}$, $\deg(R\circ S)\leqq\deg(R)+\deg(S)$ and $\text{hgt}(R\circ S)\leqq 2\min\{(1+\deg(R)),\ (1+\deg(S))\}\ \text{hgt}(R)\,\text{hgt}(S)$; $\deg(R*S)\leqq\deg(R)+\deg(S)$ and $\text{hgt}(R*S)\leqq\min\{(1+\deg(R)),\ (1+\deg(S))\}\ \text{hgt}(R)\ \text{hgt}(S)$.*

Proof. The bounds follow from Lemma 2.1 and from the fact that for $\circ\in\{+,-\}$, $R\circ S=(P_1Q_2\circ Q_1P_2)/P_2Q_2$. □

For a polynomial $P(x)$, let the *leading monomial* of $P(x)$ be the monomial of maximum degree in $P(x)$. Let the *leading coefficient* of $P(x)$ be the coefficient of its leading monomial.

LEMMA 2.3. *Let $P(x)$ and $Q(x)$ be two polynomials with integer coefficients, and let L be the leading coefficient of $Q(x)$. Then, $P(x)=(A(x)Q(x)/L^{\delta+1})+(R(x)/L^{\delta+1})$, where $A(x)$ and $R(x)$ are polynomials with integer coefficients, $\text{hgt}(A)\leqq\text{hgt}(P)(2\,\text{hgt}(Q))^{\delta}$, $\text{hgt}(R)\leqq\text{hgt}(P)(2\,\text{hgt}(Q))^{\delta+1}$, $\delta=\max\{-1,\deg(P)-\deg(Q)\}$, and $\deg(R)<\deg(Q)$.*

Proof. The proof is by induction on δ. The hypothesis holds for the basis case $\delta=-1$ with $A(x)=0$ and $R(x)=P(x)$.

---

[4] Throughout the paper, the product operator is omitted when no confusion may arise.

For the induction step, assume that the hypothesis holds for all $j<\delta$, for some $\delta>-1$. We prove it for $\delta=\deg(P)-\deg(Q)$. Let $\tilde{L}$ be the leading coefficient of $P(x)$. Consider the polynomial $S(x)=LP(x)-x^{\delta}\tilde{L}Q(x)$. Clearly, $\operatorname{hgt}(S)\leqq 2\operatorname{hgt}(P)\operatorname{hgt}(Q)$, $\deg(S)\leqq\deg(P)-1$.

Applying the hypothesis to the pair $S(x)$ and $Q(x)$, yields $S(x)=(A(x)Q(x)/L^{\delta})+(R(x)/L^{\delta})$. Substituting for $S(x)$, we get

$$P(x)=\frac{1}{L^{\delta+1}}(A(x)+L^{\delta}x^{\delta}\tilde{L})Q(x)+\frac{1}{L^{\delta+1}}R(x).$$

In addition, $\operatorname{hgt}(A)\leqq\operatorname{hgt}(P)(2\operatorname{hgt}(Q))^{\delta}$, $\operatorname{hgt}(R)\leqq\operatorname{hgt}(P)(2\operatorname{hgt}(Q))^{\delta+1}$, and $\deg(R)<\deg(Q)$. □

COROLLARY 2.4. *Let $P(x)$ and $Q(x)$ be two polynomials with integer coefficients of degree strictly less than a positive integer D, and height strictly less than M, and let L be the leading coefficient of $Q(x)$, then $P(x)=(A(x)Q(x)/L^{D})+(R(x)/L^{D})$, where $A(x)$ and $R(x)$ are polynomials with integer coefficients,* $\operatorname{hgt}(A)<2^{D-1}M^{D}$, $\operatorname{hgt}(R)<2^{D}M^{D+1}$, *and* $\deg(R)<\deg(Q)$.

2.3. A lexicographic order on the polynomials. We define a lexicographic order on the set of univariate polynomials. For this purpose, we use the following lexicographic order on the set of monomials.

DEFINITION 1. For two monomials cx^{i} and dx^{j}, $cx^{i}>dx^{j}$ if either (1) $i>j$ or (2) $i=j$ and $|c|>|d|$.

We say that a univariate polynomial is written in its *normal form* if it is written as a minimal sum of monomials, and these monomials are sorted in descending lexicographic order. Throughout this paper, we assume that all polynomials are written in their normal form.

DEFINITION 2. For two polynomials $P(x)$ and $Q(x)$, $P(x)>Q(x)$ if, when written in their normal forms, there exists some $i\geqq 1$, such that (the ith monomial in P) > (the ith monomial in Q), and all the monomials preceding it are identical in both P and Q.

Below, we relate the lexicographic order defined on the polynomials, and the order among their values at certain points.

LEMMA 2.5. *For each polynomial $P(x)$, there exists a positive integer $\pi(P)$ such that for all $a>\pi(P)$ the* sign *of $P(a)$ is the same as the sign of the leading coefficient of P. Furthermore,* $\pi(P)\leqq\operatorname{hgt}(P)/|L|+1$, *where $L\neq 0$ is the leading coefficient of P.*

Proof. Without loss of generality we may assume that $L>0$. (In case $L<0$ we get the bound by considering $-P(x)$.) It is not difficult to see that the Lemma holds if we take $\pi(P)$ to be the value of the largest real root of $P(x)$. It is well known (see, e.g., [Hou70, Thm. 2.2.4]) that every root of $P(x)$ is bounded from above by $\operatorname{hgt}(P)/L+1$. □

From Lemma 2.5 it follows that for any two polynomials $Q(x)$ and $R(x)$, there exists a positive integer $\pi(Q-R)$ such that for all $a>\pi(Q-R)$, either always $Q(a)<R(a)$ or always $Q(a)\geqq R(a)$.

3. A lower bound for computation trees. In this section, we prove an $\Omega(\sqrt{\log\log M(n)})$ lower bound on the depth of any computation tree with $OP=\{+,-,*,/,\lfloor\cdot\rfloor\}$, that computes an $M(n)$-invariant function, for all n-bit integers. We assume that "0" and "1" are the only constants explicitly involved in any operation performed in the tree (and that any other constant used in the tree must be computed). If we drop this assumption (i.e., when the set of rational numbers is the set of allowable constants), then the following theorem implies that there is no finite depth computation

tree with $OP=\{+,-,*,/,\lfloor\cdot\rfloor\}$, that computes an $\omega(1)$-invariant function, for all integers.

First, we prove the following key lemma.

DEFINITION 3. Let the set $S(\lambda)$ be the set of all n-bit integers that are multiples of λ.

LEMMA 3.1. *Let T be any computation tree for a one input problem of depth h with* $OP=\{+,-,*,/,\lfloor\cdot\rfloor\}$. *Then, there is an integer* $\lambda<2^{2^{4h^2}}$ *and a leaf* v_l *of T, such that for each input* $a\in S(\lambda)$, *the computation follows the path from the root of T to its leaf* v_l.

Proof of Lemma 3.1. Let $\mathcal{P}$ be the path from the root of T to the leaf v_l. Denote the vertices on the path $\mathcal{P}$ by $v_1, v_2, \cdots, v_l$, in that order, where v_1 is the root of the tree T and v_i is a child of v_{i-1}. We will define the path $\mathcal{P}$ and the integer λ inductively, starting with the path v_1, v_2 (v_1 is the input vertex and v_2 is its only child), and $\lambda^{(1)}=1$. ($S(1)$ consists of all n-bit integers.) As part of the induction hypothesis, we will maintain five properties of the path and the set under consideration. These properties are described below.

Suppose that (a) we have selected a prefix of $\mathcal{P}$, which starts at v_1, and ends at a vertex v_{i+1}, and (b) we have defined a parameter $\lambda^{(i)}$ with the following properties:

(1) For each input $a\in S(\lambda^{(i)})$ the computation follows the path from the root to v_{i+1}.

(2) For each computation vertex ν on the path from the root to the vertex v_{i+1}, excluding the vertex v_{i+1}, there is a pair of polynomials $(F_\nu(x), G_\nu(x))$ with integer coefficients, such that for each input $a\in S(\lambda^{(i)})$, $G_\nu(a)\neq 0$, and $f_\nu(a)=F_\nu(a)/G_\nu(a)$.

(3) The leading coefficient of each of the polynomials $G_\nu(x)$ is positive.

(4) For each polynomial $F_\nu(x)$ and each $a\in S(\lambda^{(i)})$, the sign of $F_\nu(a)$ is the same as the sign of the leading monomial of $F_\nu(x)$. This also holds for all polynomials $G_\nu(x)$.

(5) Let $\Sigma_i=\{F_{v_j}, G_{v_j}\mid v_j$ is a computation vertex, $j\leqq i\}$. Define $D_i=\deg(\Sigma_i)+1$ and $M_i=\mathrm{hgt}(\Sigma_i)$. Then, $D_i<2^i$, and $\max\{\lambda^{(i)}, M_i\}<2^{2^{4i^2}}$.

We show how to define $\lambda^{(i+1)}$ and how to choose the vertex v_{i+2} such that Properties 1–5 will be satisfied by the set $S(\lambda^{(i+1)})$ and the prefix of $\mathcal{P}$ that starts at v_1, and ends at v_{i+2}. The parameter $\lambda^{(i+1)}$ will be a multiple of $\lambda^{(i)}$. Notice that this implies that $S(\lambda^{(i+1)})\subseteq S(\lambda^{(i)})$, therefore, by the induction hypothesis we have that Properties 1–5 are satisfied by the set $S(\lambda^{(i+1)})$ and the prefix of $\mathcal{P}$ that starts at v_1 and ends at v_{i+1}. In order to complete the proof of the lemma we need to show that (a) there exists an outgoing edge of v_{i+1} such that for each input $a\in S(\lambda^{(i+1)})$ the computation follows this edge, and (b) Properties 2–5 are satisfied also for the vertex v_{i+1} and the set $S(\lambda^{(i+1)})$.

By the definition of the tree T, the vertex v_{i+1} is either a comparison vertex or a computation vertex. If it is a comparison vertex then a comparison $g:h$ is resolved. If it is a computation vertex then either $f_{v_{i+1}}=g\circ h$, for $\circ\in\{+,-,*,/\}$, or $f_{v_{i+1}}=\lfloor g\rfloor$ is evaluated. Here, $g,h\in\{0,1\}\cup\{f_{v_j}\mid v_j$ is a computation vertex, $j\leqq i\}$ and $\circ\in\{+,-,*,/,\lfloor\cdot\rfloor\}$. We will use the following notation in the rest of this proof.

$$(P_1(x), P_2(x))=\begin{cases}(g,1) & \text{if } g\in\{0,1\},\\ (F_\nu(x), G_\nu(x)) & \text{if } g=f_\nu,\end{cases}$$

and

$$(Q_1(x), Q_2(x))=\begin{cases}(h,1) & \text{if } h\in\{0,1\},\\ (F_\nu(x), G_\nu(x)) & \text{if } h=f_\nu.\end{cases}$$

Finally, let $P(x)=P_1(x)Q_2(x)$, $Q(x)=P_2(x)Q_1(x)$.

The proof is based on a case by case analysis. In each case, we will define the next vertex v_{i+2} on the path $\mathcal{P}$, the parameter $\lambda^{(i+1)}$, and the polynomials $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$.

First, we resolve the case when v_{i+1} is a *comparison* vertex. Let $B(x) = P(x) - Q(x) \neq 0$. Then, Lemma 2.5 guarantees the existence of a positive integer $\pi(B)$, and $\bullet \in \{<, >\}$, such that for all integers $a > \pi(B)$, $(P_1(a)/P_2(a)) \bullet (Q_1(a)/Q_2(a))$. Thus, for all integers a with this property, that arrive at the vertex v_{i+1}, the next vertex v_{i+2} is either the left child of v_{i+1} (if $\bullet$ is $<$), or the right child of v_{i+1} (if $\bullet$ is $>$). Define $\lambda^{(i+1)} = \lambda^{(i)}\pi(B)$. Recall that each of P and Q is a product of two polynomials of degree less than D_i, and height at most M_i. Therefore, $\deg(P)$, $\deg(Q) < 2D_i$ and $\operatorname{hgt}(P)$, $\operatorname{hgt}(Q) < D_i M_i^2$. Lemma 2.5 implies that $\pi(B) \leqq 2D_i M_i^2$, hence $\lambda^{(i+1)} < 2^{2^{4(i+1)^2}}$. For each $a \in S(\lambda^{(i+1)})$ Properties 1-5 are satisfied.

Next, consider the case when v_{i+1} is a computation vertex. The following possibilities may arise.

Suppose that $\circ \in \{+, -\}$. This case is very similar to the case of a comparison vertex discussed above. Following the argument in that case, let $B(x) = P(x) \circ Q(x)$. Define $\lambda^{(i+1)} = \lambda^{(i)}\pi(B)$. Let $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (B(x), P_2(x)Q_2(x))$. The degree of B is less then $2D_i$, and its height is bounded by $2D_i M_i^2$. Therefore, $D_{i+1} < 2D_i$, $M_{i+1} \leqq 2D_i M_i^2$. For each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1-5 are satisfied.

Suppose that $\circ = *$. This is the simplest case. Let $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (P_1(x)Q_1(x), P_2(x)Q_2(x))$. Define $\lambda^{(i+1)} = \lambda^{(i)}$. Clearly, for each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1-5 are satisfied.

Next, suppose that $\circ = /$. Let ρ be the *sign* of the leading coefficient of $Q_1(x)$. Define $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (\rho P(x), \rho Q(x))$ and $\lambda^{(i+1)} = \lambda^{(i)}$. It follows that $D_{i+1} < 2D_i$, $M_{i+1} \leqq D_i M_i^2$. This is the only case where $G_{v_{i+1}}(x)$ is not a product of $G_{v_j}(x)$'s, for $j \leqq i$. Nevertheless, $G_{v_{i+1}}(a) \neq 0$, for any $a \in S(\lambda^{(i+1)})$ because T is a well-defined computation tree that does not contain any division by zero. Clearly, for each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1-5 are satisfied.

The only remaining case is when $\circ =$ **floor**. The rest of this section is devoted to this case.

We may assume, without loss of generality, that the operand of any **floor** operation is nonnegative. Then, Property 3 implies that the leading coefficient of $P_1(x)$ may be assumed to be positive.

In the trivial case when $P_1(x) = P_2(x)$, we just define $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (1, 1)$. We distinguish between two nontrivial cases.

Case 1. $P_1(x) < P_2(x)$. Let $B(x) = P_2(x) - P_1(x)$. Since the leading coefficient of B is positive, Lemma 2.5 guarantees the existence of a positive integer $\pi(B)$ such that $B(a) > 0$, for all $a > \pi(B)$. This implies that $1 > P_1(a)/P_2(a) \geqq 0$, for all $a > \pi(B)$. Let $\lambda^{(i+1)} = \pi(B)\lambda^{(i)}$. We conclude that for integers $a \in S(\lambda^{(i+1)})$, $\lfloor P_1(a)/P_2(a) \rfloor = 0$. Define $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (0, 1)$. By an argument similar to that in the case where v_{i+1} is a comparison vertex, it can be shown that for each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1-5 are satisfied.

Case 2. $P_2(x) < P_1(x)$. Let L be the leading coefficient of $P_2(x)$. Corollary 2.4 implies that $P_1(x) = L^{-d}(A(x)P_2(x) + R(x))$, where the coefficients of $A(x)$ and $R(x)$ are integers, $d = \deg(P_1) - \deg(P_2) + 1$, and $\deg(R) < \deg(P_2)$.

Consider the constant term of $A(x)$. We denote this constant by $cL^d + \gamma$, where c is an integer and γ is a nonnegative integer such that $0 \leqq \gamma < L^d$. Let $A(x) = \hat{A}(x) + cL^d + \gamma$; that is, $\hat{A}(x)$ is equal to the polynomial $A(x)$ minus its constant term. The parameter $\lambda^{(i+1)}$ will be chosen to be a multiple of $L^d\lambda^{(i)}$. This implies that for each $a \in S(\lambda^{(i+1)})$, each monomial of $L^{-d}\hat{A}(a)$ evaluates to an integer. Hence, for each

such integer a,

$$\left\lfloor \frac{P_1(a)}{P_2(a)} \right\rfloor = L^{-d}\hat{A}(a) + c + \left\lfloor \frac{\gamma P_2(a) + R(a)}{L^d P_2(a)} \right\rfloor.$$

We distinguish between two subcases.

Subcase 1. $\gamma > 0$. Consider the polynomial $L^{-d}(\gamma P_2(x) + R(x))$. Since $\deg(R) < \deg(P_2)$ the leading coefficient of this polynomial is $L^{-d}\gamma$ times the leading coefficient of $P_2(x)$ (which is L). Let $B(x) = P_2(x) - L^{-d}(\gamma P_2(x) + R(x))$. The leading coefficient of $B(x)$ is $(1 - L^{-d}\gamma)L$. Since $0 < L^{-d}\gamma < 1$, the leading coefficient of $B(x)$ is positive. Using Lemma 2.5, let π be the minimum multiple of L^d such that $\pi \geqq \max\{\pi(B), \pi(\gamma P_2 + R)\}$. Then, $0 < L^{-d}(\gamma P_2(a) + R(a)) < P_2(a)$, for all $a > \pi$. This implies that $0 < (\gamma P_2(a) + R(a))/L^d P_2(a) < 1$, for all $a > \pi$. We conclude that for each such a, $\lfloor P_1(a)/P_2(a) \rfloor = L^{-d}\hat{A}(a) + c$. Define $\lambda^{(i+1)} = \pi\lambda^{(i)}$, and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (\hat{A}(x) + cL^d, L^d)$. For each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1–4 are satisfied.

Below, we prove that Property 5 is also satisfied. To get the bound on D_{i+1} and M_{i+1} it is sufficient to consider the new polynomial $L^{-d}\hat{A}(x) + c$. Clearly, $\deg(L^{-d}\hat{A}(x) + c) < D_i$. The height of the polynomial can be bounded using Corollary 2.4 as follows:

$$\operatorname{hgt}(\hat{A}(x) + cL^d) \leqq \operatorname{hgt}(A) < (2M_i)^{D_i} < 2^{2^{4(i+1)^2}}.$$

To get the bound on $\lambda^{(i+1)}$, recall that $L^d B(x) = L^d P_2(x) - (\gamma P_2(x) + R(x))$. Therefore, $\deg(L^d B) < D_i$, and $\operatorname{hgt}(L^d B) \leqq L^d \operatorname{hgt}(P_2) + \operatorname{hgt}(R) < M_i^{D_i} + 2^{D_i} M_i^{D_i+1}$. Since $L^d B$ is a polynomial with integer coefficients also $\pi(B) \leqq \operatorname{hgt}(L^d B) + 1$. This implies that $\lambda^{(i+1)} < \lambda^{(i)}(2M_i)^{D_i+1} < 2^{2^{4(i+1)^2}}$.

Subcase 2. $\gamma = 0$. Clearly $R(x) < L^d P_2(x)$. Using Lemma 2.5, let π be the minimum multiple of L^d such that $\pi \geqq \max\{\pi(R), \pi(L^d P_2 - R)\}$. Then, for all $a > \pi$, $0 \leqq R(a)/L^d P_2(a) < 1$ if (leading coefficient of $R(x)) \geqq 0$, and $-1 < R(a)/L^d P_2(a) < 0$ otherwise. Let $\tilde{c}$ be c if the leading coefficient of $R(x) \geqq 0$, and $c-1$ otherwise. We conclude that for each such a, $\lfloor P_1(a)/P_2(a) \rfloor = L^{-d}\hat{A}(a) + \tilde{c}$. Define $\lambda^{(i+1)} = \pi\lambda^{(i)}$, and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x)) = (\hat{A}(x) + \tilde{c}L^d, L^d)$. For each $a \in S(\lambda^{(i+1)})$ and $(F_{v_{i+1}}(x), G_{v_{i+1}}(x))$, Properties 1–4 are satisfied.

Below, we prove that Property 5 is also satisfied. To get the bound on D_{i+1} and M_{i+1} it is sufficient to consider the new polynomial $L^{-d}\hat{A}(x) + \tilde{c}$. To get the bound on $\lambda^{(i+1)}$ we have to consider also the polynomial $L^d P_2(x) - R(x)$. By an argument similar to that in Subcase 1, Property 5 holds. □

Lemma 3.1 readily implies the following theorem.

THEOREM 3.2. *Let $f(\cdot)$ be an $M(n)$-invariant function. Then any computation tree with $OP = \{+, -, *, /, \lfloor \cdot \rfloor\}$ and constants $\{0, 1\}$, that computes $f(\cdot)$, for all n-bit integers, has depth $\Omega(\sqrt{\log\log M(n)})$.*

Proof. We prove the lower bound by contradiction. Suppose that there is a computation tree T of depth $h < \frac{1}{2}\sqrt{\log\log M(n)}$, that computes $f(\cdot)$ for all n-bit integers. By Lemma 3.1 there exists a $\lambda < M(n)$ such that the computation for each input $a \in S(\lambda)$ follows the same path in T.

Since $f(\cdot)$ is an $M(n)$-invariant function, there are two inputs $a_0, a_1 \in S(\lambda)$, such that $f(a_0) \neq f(a_1)$. Since a_0 and a_1 follow the same path in T, T produces the same output for both of them. However, for either a_0 or a_1 the value is incorrect—a contradiction. □

COROLLARY 3.3. *Let $f(\cdot)$ be an $M(n)$-invariant function. If $M(n) = \Omega(2^{n^\varepsilon})$, for a fixed ε, then any computation tree with $OP = \{+, -, *, /, \lfloor \cdot \rfloor\}$ and constants $\{0, 1\}$, that computes $f(x)$ for all n-bit integers must have depth $\Omega(\sqrt{\log n})$.*

Corollary 3.3 implies an $\Omega(\sqrt{\log n})$ lower bound on the depth of any computation tree with $OP=\{+,-,*,/,\lfloor\cdot\rfloor\}$, that solves the following problems.

(1) Decide if $\lfloor\log a\rfloor$ is odd or even, for any n-bit integer a. (Let $M(n)=2^{n-1}$, $a_0=\lambda$, and $a_1=2\lambda$.)

(2) Decide if $\lfloor\log\log a\rfloor$ is odd or even, for any n-bit integer a. (Let $M(n)=2^{n/2}$, $a_0=\lambda$, and $a_1=\lambda^2$.)

(3) Decide if an n-bit integer a is a Perfect Square, i.e., decide if $\sqrt{a}$ is an integer. (Let $M(n)=2^{n/2-1}$, $a_0=\lambda^2$, and $a_1=2\lambda^2$.)

We remark that the lower bounds apply also to the computation problems that correspond to each of the above decision problems; this is because these computation problems are at least as hard as the decision problems.

4. A lower bound for RAMs. In this section, we extend the techniques of the previous sections in order to obtain an $\Omega(\sqrt{\log\log M(n)})$ lower bound on the time complexity of any RAM program with operations $\{+,-,*,/,\lfloor\cdot\rfloor\}$ that computes an $M(n)$-invariant function for all n-bit integers. As discussed in the following paragraphs, the main issue in this section is the indirect addressing capability of the RAM.

Consider a RAM program $\mathcal{R}$ of time complexity h, without indirect addressing. All computations of $\mathcal{R}$ can be simulated by a computation tree $T_h^{\mathcal{R}}$ of depth $O(h)$, in a straightforward manner. On the other hand, in the case when $\mathcal{R}$ is capable of indirect addressing, no straightforward simulation by a computation tree of depth $O(h)$ appears possible. (In [Mey89], it is established that a RAM program $\mathcal{R}$ of time complexity h with indirect addressing can be simulated by a computation tree of depth $O(h\log h)$.) Paul and Simon [PS81] show that a simulation by a computation tree of depth $O(h)$ is possible, provided one restricts the set of inputs in a suitable manner. However, this simulation also depends on the fact that the set of instructions does not include the **floor** operation, but consists only of $\{+,-,*,/\}$.

We assume that the reader is familiar with the arguments in [PS81]. The main idea in that paper is to restrict the set of inputs so that no pair of distinct polynomials used to compute addresses (for indirect addressing) evaluate to the same address value on any input in the restricted set.

In the proof of the following theorem, we combine the ideas of [PS81] with the arguments of the previous sections.

THEOREM 4.1. *Any* RAM *program with* $OP=\{+,-,*,/,\lfloor\cdot\rfloor\}$, *that computes an* $M(n)$*-invariant function for all n-bit integers has time complexity* $\Omega(\sqrt{\log\log M(n)})$.

The proof of this theorem is the same as the proof of Theorem 3.2, except that the use of Lemma 3.1 in that proof is replaced by the following lemma.

LEMMA 4.2. *Let* $\mathcal{R}$ *be a* RAM *program for a one input problem of time complexity* h *with operations* $\{+,-,*,/,\lfloor\cdot\rfloor\}$ *and program* $((1:\gamma_1),(2:\gamma_2),\cdots,(r:\gamma_r))$. *Then, there is a* $\lambda<2^{2^{4h^2}}$ *such that the same sequence of instructions is executed for all inputs from* $S(\lambda)$.

Proof. The proof is very similar to the proof of Lemma 3.1. The main additional effort is in establishing a correspondence between a set of rational functions and the set of memory locations accessed by the RAM program on any given input. This idea appears in the work of Paul and Simon [PS81].

Let us denote by $\mathcal{P}=\gamma_{j_1},\gamma_{j_2},\cdots,\gamma_{j_\omega}$ the sequence of instructions asserted in the statement of the lemma. We will show that there exists a $\lambda<2^{2^{4h^2}}$ such that $\mathcal{P}$ is the sequence of instructions executed for all inputs from $S(\lambda)$, and γ_{j_ω} is the **halt** instruction.

As in the proof of Lemma 3.1, the sequence $\mathcal{P}$ and the integer λ are defined inductively, starting with the empty sequence and the set $\lambda^{(0)}=1$. ($S(1)$ consists of all n-bit integers.) In addition, define $A^{(i)}$ to be a set of rational functions such that the

set of memory locations accessed by the RAM program in the first i steps, on input $a \in S(\lambda^{(i)})$, is a subset of $\{F(a)/G(a) | F(x)/G(x) \in A^{(i)}\}$. To begin with, define $A^{(0)}$ to be the empty set, and let $j_1 = 1$.

Following that proof, suppose that we have (i) selected a prefix of $\mathcal{P}$, which starts at γ_1 and ends at $\gamma_{j_{i+1}}$; (ii) defined $\lambda^{(i)}$, and therefore, constructed the set $S(\lambda^{(i)})$ of inputs; and, in addition, (iii) defined a set $A^{(i)}$. Furthermore, assume that Properties 1–5 of Lemma 3.1 are satisfied, and in addition, the following sixth property is also satisfied:

(6) If $r(x)$ and $s(x)$ are two distinct rational functions in $A^{(i)}$, then $r(a) \neq s(a)$, for any $a \in S(\lambda^{(i)})$.

Property 6 implies that for any input $a \in S(\lambda^{(i)})$, all rational functions in the set $A^{(i)}$ evaluate to different values. Therefore, for each input $a \in S^{(i)}$, we may set up a one to one correspondence between the rational functions in the set $A^{(i)}$, and the set of memory locations accessed by the RAM program.

Suppose that all the operands of the instruction $\gamma_{j_{i+1}}$ have been identified. Then, as in the proof of Lemma 3.1, we define $\lambda^{(i+1)}$, which is a multiple of $\lambda^{(i)}$ such that Properties 1–5 are also satisfied for the instruction $\gamma_{j_{i+1}}$ and each input $a \in S(\lambda^{(i+1)})$. If $\gamma_{j_{i+1}}$ is not the **halt** instruction, then we also select the next instruction $\gamma_{j_{i+2}}$ in the sequence $\mathcal{P}$. In the following, we show how to identify the operands of $\gamma_{j_{i+1}}$, define $A^{(i+1)}$, and prove that Property 6 can also be maintained in this construction.

Initialize $A^{(i+1)}$ to $A^{(i)}$. Suppose that the value in the memory location t, denoted by $MEM(t)$, is an operand for the instruction $\gamma_{j_{i+1}}$. By the definition of the RAM, the following two cases may arise:

(1) Direct Addressing, i.e., t is an integer constant.

(2) Indirect Addressing, i.e., $t = MEM(m)$. Without loss of generality, we may assume that m is an integer.

Next, we identify a rational function, denoted by $w(x)$, that must be included in the set $A^{(i+1)}$.

In the first case, include $w(x) = t$ (a constant polynomial) in the set $A^{(i+1)}$.

In the second case, let k be the largest index such that the instruction γ_{j_k} assigns a value to the location $MEM(m)$. By the induction hypothesis, this value (which equals t) is the value of the rational function $F_k(x)/G_k(x)$, evaluated at the input $a \in S(\lambda^{(i)})$. Include $w(x) = F_k(x)/G_k(x)$ in the set $A^{(i+1)}$.

To identify the operand of $\gamma_{j_{i+1}}$, corresponding to $MEM(t)$, let l be the largest index such that the instruction γ_{j_l} assigns a value to the location with address $w(x)$. Then, the operand of $\gamma_{j_{i+1}}$, corresponding to $MEM(t)$, is given by $F_l(x)/G_l(x)$.

In this manner, we identify all the operands of the instruction $\gamma_{j_{i+1}}$.

In a similar manner, include in $A^{(i+1)}$ the rational function corresponding to the address of the location where the result of the instruction $\gamma_{j_{i+1}}$ is stored.

In order to maintain Property 6, we only need to ensure that $(r(x) - s(x))|_a \neq 0$, whenever $a \in S(\lambda^{(i+1)})$, for any $r(x), s(x) \in A^{(i+1)}$. This will guarantee that all rational functions in the set $A^{(i+1)}$ evaluate to different values at each input $a \in S(\lambda^{(i+1)})$. Implying that different rational functions in $A^{(i+1)}$ correspond to different memory locations. By Lemma 2.5, this can be achieved by updating $\lambda^{(i+1)}$. The same lemma also implies that such updates can be performed without violating Property 5. □

5. The $O(\log n)$ upper bound for $OP = \{+, -, *, /, [\cdot]\}$. For completeness, we briefly recall how a RAM (without indirect addressing) can solve the Perfect Square Problem for n-bit inputs in $O(\log n)$ steps. This is a well-known algorithm [Ral65] based on the Newton Iteration for computing the square root of a number.

Suppose that the input integer is a, $0 < a < 2^n$. The first step is to determine k such that $2^{2k} \leqq a < 2^{2k+2}$. This is described in the following paragraph. Recall that '1' is the only constant allowed in the computation. Therefore, we must compute all the other constants needed in the algorithm.

Use repeated squaring to compute numbers 2^{2^i}, $i = 1, 2, \cdots, m$, such that m is the least integer for which $2^{2^m} \geqq a$. Clearly, $m \leqq \lceil \log n \rceil$. Finally, use these numbers and determine k by a binary search for the leading bit of a.

The next step is to compute α such that $a = 2^{2k}(1+\alpha)$. Now apply the following Newton Iteration:

$$x_{i+1} = \frac{a + x_i^2}{2x_i} \text{ starting with } x_0 = 2^k\left(1 + \frac{\alpha}{2}\right).$$

If we denote the error by $\varepsilon_i = x_i^2 - a$, then it is easy to verify that $\varepsilon_{i+1} = (\varepsilon_i/2x_i)^2$. It follows that for some $i = O(\log n)$ iterations, $0 \leqq x_i^2 - a \leqq 1$. If a is a perfect square, then its square root is $\lfloor x_i \rfloor$, and this can be tested easily.

6. Conclusion. We have proved an $\Omega(\sqrt{\log \log M(n)})$ lower bound on the depth of any computation tree with operations from the set $\{+, -, *, /, \lfloor \cdot \rfloor\}$, that computes an $M(n)$-invariant function, for all n-bit integers. We then extended the arguments to obtain the same lower bound on the time complexity of any RAM program with the operations $\{+, -, *, /, \lfloor \cdot \rfloor\}$ that computes this function. We claim that many functions are $M(n)$-invariant for a suitable choice of $M(n)$ and give some examples.

For all $M(n)$-invariant functions considered in the paper our lower bound is not tight. It would be interesting to find a specific $M(n)$-invariant function for which this bound is tight, or to achieve tight lower bounds for the functions considered here.

Finally, see [MST88a] for a discussion of other related problems.

Acknowledgments. We are grateful to Michael Ben-Or for stimulating discussions. We also wish to thank Tom Leighton for suggesting the Perfect Square Problem, and the referee for several suggestions that improved the manuscript.

Note added in proof. In a subsequent paper [MST89a], we prove that there is a computation tree of depth $O(\sqrt{r})$ with operations $\{+, -, *, /, \lfloor \cdot \rfloor, <\}$ and constants $\{0, 1\}$ that computes $\sqrt{b}$ to accuracy of one, for all integers b in the range $[2^{2^r-2}, 2^{2^r} - 1]$. Combining this upper bound with the lower bound in this paper yields an $O(\sqrt{r})$ tight bound for this problem.

REFERENCES

[AHU74] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Ben83] M. BEN-OR, *Lower bounds for algebraic computation trees*, in Proc. 15th ACM Symposium on Theory of Computing, Boston, MA, May 1983, pp. 80–86.

[BJM88] L. BABAI, B. JUST, AND F. MEYER AUF DER HEIDE, *On the limits of computations with the floor function*, Inform. and Comput., 78 (1988), pp. 99–107.

[Cas71] J. W. S. CASSELS, *An Introduction to the Geometry of Numbers*, Second Edition, Springer-Verlag, Berlin, Heidelberg, New York, 1971.

[Cob66] A. COBHAM, *The recognition problem for the set of perfect squares*, Tech. Report RC 1704, IBM T. J. Watson Research Center, Yorktown Heights, NY, April 1966.

[DO85] E. DITTERT AND M. O'DONNELL, *Lower bounds for sorting with realistic instruction sets*, IEEE Trans. Comput., 34 (1985), pp. 311–317; See also, *Correction to: Lower bounds for sorting with realistic instruction sets*, IEEE Trans. Comput., 35 (1986), p. 932.

[GLS88] M. GRÖTSCHEL, L. LOVÁSZ, AND A. SCHRIJVER, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin, New York, 1988.

[Hou70] A. S. HOUSEHOLDER, *The Numerical Treatment of a Single Nonlinear Equation*, McGraw-Hill, New York, 1970.

[IMR83] O. H. IBARRA, S. MORAN, AND L. E. ROSIER, *On the control power of integer division*, Theoret. Comput. Sci., 24 (1983), pp. 35-52.

[JMW89] B. JUST, F. MEYER AUF DER HEIDE, AND A. WIGDERSON, *On computations with integer division*, RAIRO Inform. Theoret. Appl., 23 (1989), pp. 101-111.

[Kun73] H. T. KUNG, *A bound on the multiplicative efficiency of iteration*, J. Comput. Systems Sci., 7 (1973), pp. 334-342.

[LM85] C. LAUTEMANN AND F. MEYER AUF DER HEIDE, *Lower time bounds for integer programming with two variables*, Inform. Process. Lett., 21 (1985), pp. 101-105.

[Mey89] F. MEYER AUF DER HEIDE, *On genuinely time bounded computations*, in Proc. 6th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 349, Paderborn, Germany, February 1989, Springer-Verlag, Berlin, New York, pp. 1-16.

[MST88a] Y. MANSOUR, B. SCHIEBER, AND P. TIWARI, *A lower bound for integer greatest common divisor computations*, Tech. Report RC 14271, IBM T. J. Watson Research Center, Yorktown Heights, NY, December 1988.

[MST88b] ———, *Lower bounds for integer greatest common divisor computations*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, October 1988, pp. 51-63; also in: Tech. Report TR 14271, IBM T. J. Watson Research Center, Yorktown Heights, NY, December 1988 and J. Assoc. Comput. Mach., to appear.

[MST89] ———, *Lower bounds for computations with the floor operation*, in Proc. 16th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 372, July 1989, Springer-Verlag, Berlin, New York, pp. 559-573; also in: Tech. Report TR 14272, IBM T. J. Watson Research Center, Yorktown Heights, NY, December 1988.

[MST89a] Y. MANSOUR, B. SCHIEBER AND P. TIWARI, *The complexity of approximating the square-root*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, October 1989, pp. 325-330.

[Pat72] M. S. PATERSON, *Efficient iterations for algebraic numbers*, in Proc. Symposium on the Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Yorktown Heights, NY, 1972, pp. 41-52.

[PS81] W. PAUL AND J. SIMON, *Decision trees and random access machines*, in Monograph. Enseign. Math. 30, 1981, pp. 331-340.

[Ral65] A. RALSTON, *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1965.

[Sch79] A. SCHÖNHAGE, *On the power of random access machines*, in Proc. 6th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 71, Graz, Austria, July 1979, Springer-Verlag, Berlin, New York, pp. 520-529.

[Sch82] A. SCHMITT, *On the computational power of the floor function*, Inform. Process. Lett., 14 (1982), pp. 1-3.

[Sim81] J. SIMON, *Division in idealized unit cost* RAMs, J. Comput. System Sci., 22 (1981), pp. 421-441.

[Sto76] L. STOCKMEYER, *Arithmetic versus boolean operations in idealized register machines*, Tech. Report RC 5954, IBM T. J. Watson Research Center, Yorktown Heights, NY, April 1976.

[Str72] V. STRASSEN, *Berechnung und Programm* I, Acta Informatica, 1 (1972), pp. 320-335.

[Str83] ———, *The computational complexity of continued fractions*, SIAM J. Comput., 12 (1983), pp. 1-27.

[SY82] J. M. STEELE AND A. C. YAO, *Lower bounds for algebraic decision trees*, J. Algorithms, 3 (1982), pp. 1-8.

[Yao89] A. C. YAO, *Lower bounds for algebraic computation trees with integer inputs*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, October 1989, pp. 308-313.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 328-351, April 1991

© 1991 Society for Industrial and Applied Mathematics

010

PROBABLY APPROXIMATE LEARNING OF SETS AND FUNCTIONS*

B. K. NATARAJAN†

Abstract. Probably approximate learning is the inference of a set (also called a concept) from a small number of sample points, under the probabilistic measure of success proposed in [L. G. Valiant, *Proc. ACM Symposium on Theory of Computing*, 1984, pp. 436-445]. Within this model, necessary and sufficient conditions are identified for efficient learning of a class of concepts defined on the strings of the binary alphabet. These results are with respect to both time and information complexity measures and are in terms of the asymptotic behaviour of the Vapnik-Chervonenkis dimension of the class. Corresponding results are also obtained for the case when the error in the learning process is to be one-sided, in that the inferred concept is required to be a subset of the concept to be learned. The scope of the learning model is then widened to include the inference of functions. The Vapnik-Chervonenkis dimension is also extended to obtain a measure called the "generalized dimension" of a class of functions. Using this measure, necessary and sufficient conditions for efficient learning of classes of functions are identified. These results are obtained for functions defined on the strings of the binary alphabet as well as for functions defined on the continuous domain of the real numbers.

Key words. probabilistic inference, sets, functions

AMS(MOS) subject classifications. 68T05, 62G99

1. Introduction.

This paper concerns algorithms that learn sets and functions from examples for them. The motivation behind the study is a need to better understand the space of problems known as "concept learning problems" in the Artificial Intelligence literature.

What follows is a brief definition of concept learning. Let Σ be the $\{0, 1\}$ alphabet and Σ^* the set of all strings of finite length on Σ. Let f denote a subset of Σ^* and F a set of such subsets. We call f a concept and F a class of concepts. We use the term concept instead of set to conform to the artificial intelligence literature. An example for f is a pair (x, y), $x \in \Sigma^*$, $y \in \{0, 1\}$, such that $x \in f$ if and only if $y = 1$. A learning algorithm for F is an algorithm that does the following: given a sufficiently large number of randomly chosen examples for any $f \in F$, the algorithm identifies $g \in F$, such that g is a good approximation of f. (These notions will be made formal later.) The aim of this paper is to study the relationship between the properties of F and the number of examples necessary and sufficient for a learning algorithm.

Let us place this paper in perspective. There are numerous papers on the concept learning problem in the artificial intelligence literature. See Michalski, Mitchell, and Carbonell [14] for a broad review. Much of this work is not formal in approach. On the other hand, many formal studies of related problems have been reported in the inductive inference literature. See Angluin and Smith [1] for an overview. Unfortunately, the basic assumptions of inductive inference turned out to be too abstract to permit the work significant practical import. More recently, Valiant [24] introduced a new formal framework for the problem, with a view towards probabilistic performance analysis. The framework appears to be of both theoretical and practical interest and the results of this paper are based on it and its variants. Related results appear in [1], [22], [4], [13], [5]-[8], and [11], amongst others. There is some overlap of results between Blumer et al. [8] and this paper. Specifically, Blumer et al. [8] obtain results on the learnability of concepts on general domains while we obtain results that hold

* Received by the editors June 21, 1989; accepted for publication (in revised form) July 13, 1990. Some of the work presented here was performed at the Robotics Institute of Carnegie Mellon University.

† Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304.

for concepts on discrete domains such as the strings over a finite alphabet. Our results are significant for two reasons: first, many domains of interest are discrete, and second, our proof techniques are much simpler than those used by Blumer et al. [8].

We begin by describing a formal model of learning, a variant of the model first presented by [24]. Specifically, in § 2 we define the notions of probably approximate learning and polynomial-sample learnability for classes of concepts on Σ^*. We then obtain necessary and sufficient conditions for polynomial-sample learnability, in terms of the Vapnik–Chervonenkis dimension of a class of concepts. Section 2.1 deals with a slightly different learning model, one in which the learner is required to learn with one-sided error, i.e., the concept output by the learner must be conservative in that it is a subset of the concept to be learned. Section 3 deals with the time complexity of concept learning, identifying necessary and sufficient conditions for efficient learning. Section 4 generalizes the learning model to functions from Σ^* to Σ^*. We define the notion of polynomial-sample learnability in this setting and introduce a generalization of the Vapnik–Chervonenkis dimension of classes of functions. We then obtain necessary and sufficient conditions for polynomial-sample learnability in terms of the generalized dimension of a class of functions. Finally, § 5 considers the learnability of spaces of functions on the reals. We identify conditions necessary and sufficient for such learnability.

2. Learning concepts on discrete domains. We begin by describing our variant of the learning framework proposed by [24]. A *concept* f is any subset of Σ^*. Associated with each concept f is the *indicator function* $\mathrm{In}_f : \Sigma^* \to \{0, 1\}$ such that $\mathrm{In}_f(x) = 1$ if and only if $x \in f$. In the interest of simplicity, we will use f to denote both the set f and the indicator function In_f, relying on the context for clarity. An *example* for concept f is a pair (x, y), $x \in \Sigma^*$, $y \in \{0, 1\}$ such that $y = f(x)$. The length of an example (x, y) is $|x| + |y| = |x| + 1$.

Notation. For a string x, $|x|$ is the string length, while for a set S, $|S|$ is the cardinality.

A *class* of concepts F is any set of concepts on Σ^*. A learning algorithm for F is an algorithm that attempts to infer an approximation to a concept f in F from examples for it. The concept $f \in F$ that is to be approximated (or learned) is called the *target concept.* The learning algorithm takes three parameters as input: the error parameter ε, the confidence parameter δ, and the length parameter n. The parameters ε and δ are real numbers in the half-open interval $(0, 1]$, while n is a natural number. The significance of these parameters will be discussed shortly. The learning algorithm has at its disposal a subroutine EXAMPLE, which at each call returns a randomly chosen example for the target concept. The example is chosen randomly according to an arbitrary and unknown probability distribution P on $\Sigma^{\leqq n}$, in that the probability that a particular example $(x, f(x))$ will be produced at any call of EXAMPLE is $P(x)$.

Notation. $\mathbf{N}$ denotes the set of natural numbers. For $n \in \mathbf{N}$, $\Sigma^{\leqq n}$ is the set of all strings of length at most n in Σ^*.

With these supporting definitions in hand, we present our main definition.

DEFINITION 1. Algorithm A is a *probably approximate* learning algorithm for F if:

(1) A takes as input reals ε, $\delta \in (0, 1]$, and $n \in \mathbf{N}$.

(2) A may call EXAMPLE, which returns examples for some $f \in F$. The examples are chosen randomly and independently according to an arbitrary and unknown probability distribution P on $\Sigma^{\leqq n}$.

(3) For all concepts $f \in F$ and all probability distributions P on $\Sigma^{\leqq n}$, with probability at least $(1 - \delta)$, A outputs a concept $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

Notation. For any two sets f and g, $f\Delta g$ is the symmetric difference between f and g, i.e., $f\Delta g=(f-g)\cup(g-f)$. Also, $P(f)$ denotes the weight of the distribution P on f, i.e., $P(f)=\Sigma_{x\in f}P(x)$.

In the above, the learning algorithm is required to output a concept g that approximates the target concept to within ε. That is, $P(f\Delta g)\leqq\varepsilon$. Since ε controls the error in the output of the algorithm, we call it the error parameter. The confidence of the learning algorithm that its output is within the allowed error is at least $(1-\delta)$. Hence, we call δ the confidence parameter. Thus, the output of the learning algorithm is *probably* a good *approximation* of the target concept. This explains the title of this paper.

Note that as yet we have placed no restrictions on the kind of procedures we will permit as learning algorithms. When we speak of the output of the algorithm, we do not imply that this output is finitely presentable. In fact, we use the term "algorithm" in a very broad sense, with the only restriction being that of statistical measurability given below. In placing this restriction, it is convenient to define the notion of the learning operator Ψ associated with a learning algorithm A, by replacing the calls to EXAMPLE with a sequence of examples provided as input. In words, Ψ takes as input ε, δ, n, and C, where C is a sequence of examples. Then, Ψ runs A on inputs ε, δ, and n. When A calls EXAMPLE, Ψ gives A successive examples from C, rather than allowing A to obtain random examples from EXAMPLE. If the number of examples demanded by A exceeds the number of examples in C, $\Psi(\varepsilon, \delta, n, C)$ is undefined. Else, Ψ outputs the concept output by A.

DEFINITION 2. The *learning operator* associated with a learning algorithm A is given by:

Learning Operator Ψ
input
    $\varepsilon,\ \delta\in(0,1],\ n\in\mathbf{N}$
    C: sequence of examples.
begin
    Let $C=(x_1,y_1),(x_2,y_2),\cdots$
    Run A on inputs ε, δ, n.
    At the ith call of EXAMPLE by A
        if C has fewer than i items, Ψ is undefined.
        else give A the example (x_i, y_i).
    Output A's output.
end

We will require that the learning algorithm A be such that certain statistical properties of Ψ are well defined—we require that Ψ computes a random function. Specifically, for each set of inputs ε, δ, n, and C, and each output x, the probability that $\Psi(\varepsilon, \delta, n, C)$ will output x should be defined. We say that A is *admissible* if it satisfies this property. Note that if A is deterministic, Ψ computes a function, and hence A is definitely admissible. It is easy to see that even if A tosses coins and uses the results in its computations, A would still be admissible. But if A were nondeterministic, A may or may not be admissible. Henceforth, we will only be concerned with admissible algorithms.

We seek to establish the quantitative relationship between the number of examples required for learning and the properties of the class of concepts in question. The next step toward this aim is to set up a complexity measure for learning algorithms, to

measure the number of examples required by the algorithm as a function of the various parameters. Specifically, we introduce the notion of the sample complexity of a learning algorithm, which is a measure of the number of examples required by a learning algorithm as a function of ε, δ, and n.

DEFINITION 3. The *sample complexity* of a learning algorithm A is the function $s: \mathbf{R} \times \mathbf{R} \times \mathbf{N} \to \mathbf{N}$ such that $s(\varepsilon, \delta, n)$ is the maximum number of calls of EXAMPLE by A, the maximum being taken over all runs of A on inputs ε, δ, and n.

Notation. $\mathbf{R}$ is the set of reals.

DEFINITION 4. A class F is said to be *polynomial-sample learnable* if there exists a learning algorithm for F with sample complexity $O(p(1/\varepsilon, 1/\delta, n))$, where p is a polynomial function of its arguments.

The introduction of n as a parameter in the sample complexity merits some discussion. Specifically, we require n to be provided as input to the learning algorithm and that the probability distribution be on $\Sigma^{\leqq n}$. We then define the sample complexity as above. Alternatively, we could omit n as input to the algorithm, and define a sample complexity $\hat{s}(\varepsilon, \delta, n)$, such that on any run in which the inputs are ε and δ, and the length of the longest example seen is n, the algorithm calls EXAMPLE at most $\hat{s}(\varepsilon, \delta, n)$ times. While this alternative appears to be more "natural," it leads to some unintuitive consequences. For instance, there exist probability distributions for which the algorithm could seek infinitely many examples, while guaranteeing that at any point in its execution, the number of examples drawn is polynomially small in the length of the longest example seen up to that point. See Pitt and Warmuth [20]. To avoid such difficulties, we include n as input to the learning algorithm. It is important to note that this does not change any of the results in this paper. (Please see Appendix for details.)

We now turn our attention to a measure of complexity for a class of concepts.

DEFINITION 5. Let F be a class of concepts on a universe X. We say that F *shatters* a set $S \subseteq X$ if the set $\{f \cap S | f \in F\}$ is the power set of S.

Notation. For any set S, the power set of S is the set of all subsets of S, denoted by 2^S.

Example 1. Let S be the set $\{0, 1, 2\}$, and let F be the class of sets $\{0, 2, 4\}$, $\{0, 4\}$, $\{2\}$, $\{4\}$, $\{0, 2\}$, $\{0, 1, 4\}$, $\{1, 2\}$, $\{1, 4\}$, $\{0, 1, 1\}$. F shatters S. To see this, take any subset of S, say $S_1 = \{0, 1\}$. Then, $\{0, 1, 4\} \in F$ and $\{0, 1, 4\} \cap S = \{0, 1\}$. The same holds for any other subset S_1 of S. □

The significance of the above definition is that if F shatters a set S, then each string x in S is "independent" of the others. That is, knowing whether or not $x \in f$ tells us nothing about the membership in f of any other string in S. Thus, the cardinality of the largest set shattered by a class is a measure of the number of "degrees of freedom" possessed by it. With this in mind, we define the following.

DEFINITION 6. Let F be a class of concepts on a universe X. The *Vapnik-Chervonenkis dimension* of F, denoted by $\dim_{vc}(F)$, is the least integer d such that every set shattered by F is of cardinality at most d.

We also define an asymptotic form of the Vapnik-Chervonenkis dimension as follows.

DEFINITION 7. Let F be a class of concepts on Σ^* and let $f \in F$. The *projection* f_n of f on $\Sigma^{\leqq n}$ is the set of strings of length at most n in f, i.e., $f_n = f \cap \Sigma^{\leqq n}$. Similarly, the projection F_n of F on $\Sigma^{\leqq n}$ is given by $F_n = \{f_n | f \in F\}$.

DEFINITION 8. The *asymptotic dimension* of a class of concepts F is the function $d: \mathbf{N} \to \mathbf{N}$ such that for all n, $\dim_{vc}(F_n) = d(n)$. We denote the asymptotic dimension of F by $\dim(F)$.

For simplicity, we drop the prefix asymptotic, relying on the context for clarity. If dim (F) is $O(p(n))$ for some polynomial p, we say that F is of polynomial dimension.

The following lemma establishes a relationship between the cardinality of F_n and $\dim_{vc}(F_n)$. The lemma is a weak form of a well-known result. See Vapnik and Chervonenkis [25] or Assouad [3] for the strong form. We prove it here for completeness, as we will use a similar proof technique in a more general lemma later in the paper.

LEMMA 1 (Shattering Lemma). *Let $d = \dim_{vc}(F_n)$. Then,*

$$2^d \leqq |F_n| \leqq 2^{(n+2)d}.$$

Proof. The first inequality is immediate from the definition of shattering. Specifically, if F_n shatters a set of size d, then $|F_n| \geqq 2^d$. The second inequality is proved through the following claim.

CLAIM 1. *Let X be any finite set and let H be a set of subsets of X. If d is the cardinality of the largest set shattered by H, then*

$$|H| \leqq (|X|+1)^d.$$

Proof. By induction on $|X|$, the cardinality of X is thus proved.

Basis. The basis is clearly true for $|X| = 1$.

INDUCTION. Assume that the claim holds for $|X| = k$ and proves true for $k+1$. Let $|X| = k+1$ and let d be the cardinality of the largest set shattered by H. Pick any $x \in X$ and partition X into the two sets $\{x\}$ and $X_1 = X - \{x\}$. Define H_1 to be the set of all sets in H that are reflected about x. That is, for each set h_1 in H_1, there exists a set $h \in H$ such that h differs from h_1 only in that h does not include x. Formally,

$$H_1 = \{h_1 | h_1 \in H, \text{ there exists } h \in H, h \neq h_1 \text{ and } h_1 = h \cup \{x\}\}.$$

Let $H_2 = H - H_1$. Surely, the sets of H_2 can be distinguished on the elements of X_1. That is, no two sets of H_2 can differ only on x, by virtue of our definition of H_1. Hence, we can consider H_2 as a class of sets defined on X_1. Surely, H_2 cannot shatter a set larger than the largest set shattered by H. Hence, H_2 shatters a set of cardinality at most d. Since $|X_1| \leqq k$, by the inductive hypothesis we have $|H_2| \leqq (|X_1|+1)^d$.

Now consider H_1. Again, the sets of H_1 are all distinct on X_1. Suppose H_1 shattered a set $S \subseteq X_1$, $|S| \geqq d$. Then, H would shatter $S \cup \{x\}$. But, $|S \cup \{x\}| \geqq d+1$, which is impossible by assumption. Hence, H_1 shatters a set of at most $d-1$ elements in X_1. By the inductive hypothesis, we have

$$|H_1| \leqq (|X_1|+1)^{d-1}.$$

Combining the two bounds, we have

$$\begin{aligned} |H| &= |H - H_1| + |H_1| \\ &= |H_2| + |H_1| \leqq (|X_1|+1)^d + (|X_1|+1)^{d-1} \\ &\leqq (k+1)^d + (k+1)^{d-1} \leqq (k+2)(k+1)^{d-1} \\ &\leqq (k+2)^d \leqq (|X|+1)^d. \end{aligned}$$

Thus the claim is proved.

Returning to the lemma, we see that $X = \Sigma^{\leqq n}$ and $|X| \leqq 2^{n+1}$. Hence, if the largest set shattered by F_n is of size d,

$$|F_n| \leqq (2^{n+1}+1)^d \leqq 2^{(n+2)d}.$$

This completes the proof of the lemma. □

We need the following supporting definition before we can give the main theorem of this section.

DEFINITION 9. Let $f: X \to Y$. The set of all examples for f is denoted by $graph(f)$. That is, $graph(f) = \{(x, y) | x \in X, y = f(x)\}$. We say f is *consistent* with a set of examples S, if $S \subseteq graph(f)$.

For a concept f, we use $graph(f)$ to denote $graph(\mathrm{In}_f)$, the graph of the indicator function of f.

We can now use Lemma 1 to prove the main theorem of this section. This theorem is limited to concepts on the discrete domain of Σ^*, and hence is a special case of the general results of [8]. The significance of our theorem rests on two factors: first, many problems of interest concern the discrete domains of the strings of finite alphabets; second, for such domains, the simple proof techniques used here suffice, avoiding the heavy machinery of [25] that is used in [8].

THEOREM 1. *A class F is polynomial-sample learnable if and only if it is of polynomial dimension.*

Proof. We first show the "if" direction. The following is a learning algorithm for F.

Learning Algorithm A_1
input: ε, δ, n.
begin:
call EXAMPLE
$\frac{1}{\varepsilon}\left((n+2)\dim_{\nu c}(F_n)\ln(2)+\ln\left(\frac{1}{\delta}\right)\right)$ times.
let S be the set of examples seen.
pick a concept $g \in F$ consistent with S
and output g.
end

We now show that A_1 does indeed satisfy our requirements. Let f be the target concept. We require that with probability $(1-\delta)$, A_1 should output a concept $g \in F$, such that the $P(f \Delta g) \leqq \varepsilon$.

Let $g_n \in F_n$ be such that $P(f_n \Delta g_n) \geqq \varepsilon$. For a particular such g_n, the probability that any call of EXAMPLE will produce an example consistent with g_n is at most $(1-\varepsilon)$. Hence, the probability that m calls of EXAMPLE will produce examples all consistent with g_n is at most $(1-\varepsilon)^m$. Now, there are at most $|F_n|$ choices for g_n. Therefore, the probability that m calls of EXAMPLE will produce examples all consistent with any such choice of g_n is bounded by $|F_n|(1-\varepsilon)^m$. We will make m sufficiently large to bound this probability by δ. That is,

$$|F_n|(1-\varepsilon)^m \leqq \delta.$$

By Lemma 1,

$$|F_n| \leqq 2^{(n+2)\dim_{\nu c}(F_n)}.$$

Hence, we want

$$2^{(n+2)\dim_{\nu c}(F_n)}(1-\varepsilon)^m \leqq \delta.$$

Taking natural logarithms on both sides of the inequality, we get

$$(n+2)\dim_{\nu c}(F_n)\ln(2)+\ln(1-\varepsilon)m \leqq \ln(\delta).$$

Using the approximation $\ln(1+\alpha) \leqq \alpha$ and simplifying,

$$m \geqq \frac{1}{\varepsilon}\left((n+2)\dim_{\nu c}(F_n)\ln(2)+\ln\left(\frac{1}{\delta}\right)\right).$$

Hence, if m examples are drawn, with probability at least $(1-\delta)$, any concept $g_n \in F_n$ consistent with the examples will be such that $P(f_n \Delta g_n) \leqq \varepsilon$. But then, since P is a distribution on $\Sigma^{\leqq n}$, P is nonzero only on strings of length n or less. Thus, all the examples drawn involve strings from $\Sigma^{\leqq n}$, and g_n is consistent with these examples if and only if g is consistent with them. Also, $P(f\Delta g) = P(f_n \Delta g_n)$. It follows that, with probability at least $(1-\delta)$, any $g \in F$ that is consistent with all the examples drawn will be such that $P(f\Delta g) \leqq \varepsilon$. If $\dim_{vc}(F_n)$ is polynomial in n, the number of examples drawn is polynomial in $1/\varepsilon$, $1/\delta$, and n. Since this is the sample complexity of A, we have shown that F is polynomial-sample learnable.

This completes the proof of the "if" direction. Next we prove the "only if" direction.

Suppose F is of super-polynomial dimension and yet there exists a learning algorithm A for it with sample complexity $s(\varepsilon, \delta, n)$, for some s that is polynomial in $1/\varepsilon$, $1/\delta$, and n. For $\varepsilon < \frac{1}{4}$ and $\delta < \frac{1}{2}$, pick sufficiently large n such that $\dim_{vc}(F_n) \geqq 2s(\varepsilon, \delta, n)$. Let $d = \dim_{vc}(F_n)$ and let $m = s(\varepsilon, \delta, n)$. By definition, F_n shatters a set S such that $|S| = d$. Let x^m be an element of S^m.

Notation. Recall the notation that Σ^n is the set of all strings of length n on Σ. For sets S other than Σ, S^n is the n-fold Cartesian product of S. Each x^n in S^n is an ordered list of *n items*. For instance, $\mathbf{N}^3$ is the set of ordered triples of natural numbers.

Let P be the probability distribution that is uniform on S and zero elsewhere and let Ψ be the learning operator corresponding to A. For $f \in F$, let $f(x^m)$ denote the sequence of examples obtained from x^m. That is, if $x^m = (x_1, x_2, \cdots, x_m)$, $f(x^m)$ is the sequence of examples $(x_1, f(x_1)), (x_2, f(x_2)), \cdots, (x_m, f(x_m))$. Let $G \subseteq F$ be such that G shatters S and $|G| = 2^{|S|}$.

In what follows, we remind the reader that the output of A is not completely determined by its inputs and the examples provided to it. We only know that A is an admissible algorithm and hence we can only measure the statistical properties of its output. Let $A(\varepsilon, \delta, n)$ denote the output of A when run on input ε, δ, and n, with f as the target concept. Since A is a probably approximate learning algorithm for F, for each $f \in G$, the probability that $P(f\Delta A(\varepsilon, \delta, n)) \geqq \varepsilon$ is at most δ. That is,

$$\mathbf{P}\{P(f\Delta A(\varepsilon, \delta, n)) > \varepsilon\} \leqq \delta. \tag{1}$$

Notation. For any event E, $\mathbf{P}\{E\}$ denotes the probability of the event occurring.

Using the learning operator Ψ of A, we can write

$$\mathbf{P}\{P(f\Delta A(\varepsilon, \delta)) > \varepsilon\} = \sum_{x^m \in S^m} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) \geqq \varepsilon\}\mathbf{P}\{f(x^m)\}. \tag{2}$$

In the above, $\mathbf{P}\{f(x^m)\}$ is the probability that $f(x^m\}$ is the sequence of examples obtained by m calls of EXAMPLE.

Substituting (2) into (1) gives us,

$$\sum_{x^m \in S^m} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) > \varepsilon\}\mathbf{P}\{f(x^m)\} \leqq \delta.$$

Summing both sides of the above over all f in G, we get,

$$\sum_{f\in G} \sum_{x^m \in S^m} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) > \varepsilon\}\mathbf{P}\{f(x^m)\} \leqq \sum_{f\in G} \delta.$$

Flipping the order of the sums in the above, we get

$$\sum_{x^m \in S^m} \sum_{f\in G} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) > \varepsilon\}\mathbf{P}\{f(x^m)\} \leqq \sum_{f\in G} \delta. \tag{3}$$

We will estimate the inner sum in (3),

$$\sum_{f\in G} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) > \varepsilon\}. \tag{4}$$

It is convenient to define a switch function θ: {true, false} $\to \mathbf{N}$ as follows. For any boolean-valued predicate Q,

$$\theta(Q) = \begin{cases} 1, & \text{if } Q \text{ is true} \\ 0, & \text{otherwise.} \end{cases}$$

Using the switch function, we can expand (4) as follows.

$$\sum_{f\in G}\sum_{e\in G} \theta(P(f\Delta e) > \varepsilon)\mathbf{P}\{S\cap e = S\cap \Psi(\varepsilon, \delta, n, f(x^m))\}. \tag{5}$$

We claim that for each $f\in G$, there exists $g\in G$ such that $g(x^m) = f(x^m)$ and for all $h\in G$, at least one of the following inequalities must hold,

$$P(f\Delta h) > \varepsilon, \qquad P(g\Delta h) > \varepsilon.$$

To see this, consider the set $[x^m]$.

Notation. We use $[x^m]$ to denote the set of distinct items occurring in a sequence $x^m \in S^m$, i.e., $[x^m] = \{x\in S | x \text{ occurs in } x^m\}$.

Pick $g\in G$ such that g agrees with f on all of $[x^m]$ and disagrees with f on all of $S-[x^m]$. We denote such a choice of g by *pair*(f). Clearly, $g(x^m) = f(x^m)$. Now, h must disagree on at least half the elements in $S-[x^m]$ with either f or g. Since P is uniform on S, one of the following must hold:

$$P(f\Delta h) \geqq \frac{|S-[x^m]|}{2|S|}, \qquad P(g\Delta h) \geqq \frac{|S-[x^m]|}{2|S|}.$$

But $|[x^m]| \leqq m \leqq |S|/2$ and hence $|S-[x^m]| \geqq |S|/2$. Substituting this into the above and noting that $\varepsilon < \frac{1}{4}$ by assumption, the claim follows.

Now, split G into disjoint sets G_1 and G_2 such that for each $f\in G_1$, *pair*$(f) \in G_2$ and vice versa. Let $\zeta(f)$ denote the event $S\cap e = S\cap\Psi(\varepsilon, \delta, n, f(x^m))$. We can now write (5) as

$$\sum_{f\in G_1}\sum_{e\in G} \theta(P(f\Delta e) > \varepsilon)\mathbf{P}\{\zeta(f)\} + \sum_{g\in G_2}\sum_{e\in G} \theta(P(g\Delta e) > \varepsilon)\mathbf{P}\{\zeta(g)\}. \tag{6}$$

If $g = \mathit{pair}(f)$, then $f(x^m) = g(x^m)$. Hence, $\zeta(f) = \zeta(g)$ and we can write (6) as

$$\sum_{f\in G_1}\sum_{e\in G} (\theta(P(f\Delta e) > \varepsilon) + \theta(P(\mathit{pair}(f)\Delta e) > \varepsilon))\mathbf{P}\{\zeta(f)\}. \tag{7}$$

Since at least one of $P(f\Delta e) > \varepsilon$, or $P(\mathit{pair}(f)\Delta e) > \varepsilon$ must hold, we have

$$\theta(P(f\Delta e) > \varepsilon) + \theta(P(\mathit{pair}(f)\Delta e) > \varepsilon) \geqq 1.$$

Substituting the above into (7) and noting that (7) was derived from (4), we get

$$\begin{aligned} \sum_{f\in G} \mathbf{P}\{P(f\Delta\Psi(\varepsilon, \delta, n, f(x^m))) > \varepsilon\} &\geqq \sum_{f\in G_1}\sum_{e\in G} \mathbf{P}\{\zeta(f)\} \\ &= \sum_{f\in G_1}\sum_{e\in G} \mathbf{P}\{S\cap e = S\cap\Psi(\varepsilon, \delta, n, f(x^m))\} \\ &= \sum_{f\in G_1} 1 = \frac{1}{2}\sum_{f\in G} 1. \end{aligned} \tag{8}$$

The last equality follows from the fact that $|G_1|=|G_2|=|G|/2$. Substituting (8) into the left-hand side of (3), we get

$$\frac{1}{2}\sum_{x^m \in S^m}\sum_{f\in G}\mathbf{P}\{f(x^m)\}=\frac{1}{2}\sum_{f\in G}\sum_{x^m\in S^m}\mathbf{P}\{f(x^m)\}=\frac{1}{2}\sum_{f\in G}1. \tag{9}$$

Substituting (9) into (3), we get,

$$\frac{1}{2}\sum_{f\in G}1\leqq\sum_{f\in G}\delta,$$

which implies that $\frac{1}{2}\leqq\delta$, a contradiction since $\delta<\frac{1}{2}$. □

Example 2. Consider the class F of all axis-parallel rectangles on $\mathbf{N}\times\mathbf{N}$, i.e., rectangles with sides parallel to the axes. To be precise, we consider only those axis-parallel rectangles whose vertices have finite coordinates. For technical convenience, we include the empty set in F. Let each string x of even length in Σ^* be the binary encoding of a point (u, v) in $\mathbf{N}\times\mathbf{N}$, where $x=uv$, $|u|=|v|$. Thus, F is a class of concepts on Σ^*. Now, each rectangle is uniquely identified by either of its diagonals. Thus, the number of distinct concepts on $\Sigma^{\leqq n}$ is at most the number of distinct pairs of strings we can choose from $\Sigma^{\leqq n}$, which is at most 2^{2n+2}. Thus, $|F_n|\leqq 2^{2n+2}$ and by Lemma 1, $\dim(F)\leqq 2n+2$. By Theorem 1, F is polynomial-sample learnable.

As another example, consider the class of all regular sets on Σ^*. Since every finite set is regular, it is clear that every subset of $\Sigma^{\leqq n}$ is shattered by F_n. Thus, $\dim_{vc}(F_n)\geqq 2^n$ and hence the regular sets are not of polynomial dimension. By Theorem 2, the regular sets are not polynomial-sample learnable. □

2.1. Learning sets with one-sided error. We now consider a learning framework in which the learning algorithm is required to be conservative in its approximation—the output concept must be a subset of the target concept. We call this learning with one-sided error since the error in the approximation produced by the learner can only be to the "safe side" of the target concept, i.e., the errors can only be errors of omission. In some situations, such conservative behavior might be desirable.

DEFINITION 10. A class F is polynomial-sample learnable with *one-sided error* if F is polynomial-sample learnable by an algorithm A, such that the concept output by A is always a subset of the target concept.

Essentially, A learns with one-sided error if on any run of A, the concept g output by A is a subset of the target concept f. Thus, the errors in g are all errors of omission, in that some elements of f are missing from g. We will now obtain the analog of Theorem 1 in this setting. The following supporting definitions are required.

DEFINITION 11. Let F be a space of concepts and let $S\subseteq graph(e)$ for some $e\in F$. The *least* $g\in F$ consistent with S is such that

(1) g is consistent with S, and

(2) for all $f\in F$ consistent with S, $g\subseteq f$.

DEFINITION 12. We say that F is *minimally consistent* if for every $e\in F$ and each nonempty and finite $S\subseteq graph(e)$, there exists a least $f\in F$ consistent with S.

Example 3. The class of axis-parallel rectangles introduced in Example 2 is minimally consistent. To see this, let S be a set of examples. Let $x_{\max}$ and $x_{\min}$ be the maximum and minimum value of the x-coordinates of the positive examples in S. Similarly, let $y_{\max}$ and $y_{\min}$ be the maximum and minimum value of the y-coordinates of the positive examples in S. The axis-parallel rectangle with diagonal points $(x_{\min}, y_{\min})$, $(x_{\max}, y_{\max})$ is clearly the least such rectangle that is consistent with S. □

THEOREM 2. *A class F is polynomial-sample learnable with one-sided error if and only if F is of polynomial dimension and is minimally consistent.*

Proof. We first consider the "if" direction of the theorem. Suppose that F is of polynomial dimension and is minimally consistent. Consider the following learning algorithm for F.

Learning Algorithm A_2
input: ε, δ, n.
begin:
call EXAMPLE
$\frac{1}{\varepsilon}\left((n+2)\dim_{vc}(F_n)\ln(2)+\ln\left(\frac{1}{\delta}\right)\right)$ times.
let S be the set of examples seen.
let g be the least concept in F
consistent with S.
output g.
end

Let f be the target concept. Since g is the least concept consistent with S, surely, $g \subseteq f$. Using arguments identical to those used in the proof of Theorem 1, we can show that with probability greater than $(1-\delta)$, $P(f\Delta g) \leqq \varepsilon$.

Now for the "only if" direction of the theorem: Assume that F is polynomial-sample learnable with one-sided error by a learning algorithm A. We first show that F is minimally consistent. Let S be a nonempty and finite subset of *graph*(e) for some $e \in F$. Let P be the uniform distribution on S. Run the learning algorithm A for inputs $\varepsilon < 1/|S|$ and $\delta = \frac{1}{2}$. Since $\varepsilon < 1/|S|$, with probability $\frac{1}{2}$, A must output a concept f such that f is consistent with S. Suppose that f is not the least concept consistent with S, i.e., there exists a concept $g \in F$ consistent with S such that f is not a subset of g. Then, A does not learn with one-sided error, as g could well have been the target concept. This is a contradiction, hence f must be the least concept consistent with S. Since S is arbitrary, F is minimally consistent.

By arguments identical to those of our proof of Theorem 1, we can show that F must be of polynomial dimension. This completes the proof. □

The following results are useful in determining whether a class is minimally consistent.

We say F is closed under arbitrary intersection if for any $G \subseteq F$, $(\bigcap_{g\in G} g) \in F$. If the above holds only for G of countable (finite) cardinality, then F is said to be closed under countable (finite) intersection.

PROPOSITION 1. *Let F be a class of concepts on Σ^*. If F is closed under countable intersection, F is minimally consistent.*

Proof. Let S be a finite subset of *graph*(h) for some $h \in F$. Let S_1 be the set $\{x|(x,1)\in S\}$. Let $\{x_1, x_2, \cdots, x_i \cdots\}$ be the elements of $(\Sigma^* - S_1)$. For each x_i let $f^{(i)}$ be defined as follows: If there exists $f \in F$ such that $x_i \notin f$ and $S_1 \subseteq f$, then $f^{(i)} = f$. Else, $f^{(i)} = \Sigma^*$.

Consider

$$g = f^{(1)} \cap f^{(2)} \cap \cdots \cap f^{(i)} \cdots.$$

Since F is closed under countable intersection, $g \in F$. Also, since $S_1 \subseteq f^{(i)}$ for all i, $S_1 \subseteq g$. Suppose there exists $e \in F$ such that $S_1 \subseteq e$ and yet $g \nsubseteq e$. Then, there exists $x \in (g - S_1)$ such that $x \notin e$. Since $g \subseteq \Sigma^*$, $x \in (\Sigma^* - S_1)$ and $x \notin e$. But then, there must exist $f^{(i)}$ such that $x \notin f^i$. But this would require that $x \notin g$, a contradiction. Therefore, for all $e \in F$ such that $S_1 \subseteq e$, $g \subseteq e$. It follows that for all $e \in F$ such that $S \subseteq graph(e)$, $g \subseteq e$—hence, the proposition. □

Example 4. Returning to the axis-parallel rectangles: We claim that this class F is closed under arbitrary intersection. We begin with the observation that F is closed under finite intersection. To see this, note that the intersection of a pair of axis-parallel rectangles is also an axis-parallel rectangle. We will use this to show that F is closed under arbitrary intersection. For any $G \subseteq F$, we show that $(\bigcap_{g \in G} g) \in F$. Pick a particular $f \in G$. Let $G_1 = \{f \cap g | g \in G\}$. Surely, $(\bigcap_{g \in G} g) = (\bigcap_{g \in G_1} g)$. But, f is finite, and so G_1 is also finite. Since F is closed under finite intersection, $(\bigcap_{g \in G_1} g) \in F$. It follows that $(\bigcap_{g \in G} g) \in F$ and hence F is closed under arbitrary intersection.

By Proposition 1, the axis-parallel rectangles are minimally consistent. □

PROPOSITION 2. *There exists F that is closed under finite intersection, but that is not minimally consistent.*

Proof. For $p \in \mathbf{N}$, consider the set $f^{(p)}$ defined as follows.

$$f^{(p)} = \{0\} \cup \{1^{pi} | i \geqq 1\}.$$

Notation. 1^k is the string of k 1's.

Let F be the class $\{f^{(p)} | p \geqq 1\}$. Now, for any p, $q \geqq 1$,

$$f^{(p)} \cap f^{(q)} = f^{(r)},$$

where r is the least common multiple of p and q. Thus, F is closed under finite intersection.

Now, consider the set $S = \{(0, 1)\}$. Surely, $S \subseteq graph(f)$ for all $f \in F$. Yet we can show that there is no least $f \in F$ that is consistent with it. Suppose $f^{(p)}$ is claimed to be the least $f \in F$ consistent with S. Take $f^{(2p)}$. Surely, $S \subseteq graph(f^{(2p)})$ and $f^{(p)} \nsubseteq F^{(2p)}$. This is a contradiction—hence, the result. □

PROPOSITION 3. *There exists F that is minimally consistent but not closed under finite intersection.*

Proof. Let G be the set of all finite subsets of Σ^*. Let $f^{(i)}$ be as defined in Proposition 2, and let F be the class given by $F = \{f^{(2)}\} \cup \{f^{(3)}\} \cup G$. Since F contains all finite subsets of Σ^*, surely F is minimally consistent. But $f^{(3)} \cap f^{(2)} = f^{(6)}$ and $f^{(6)} \notin F$. Hence F is not closed under intersection. □

Let F be minimally consistent and let G be the set of all finite subsets of Σ^*. For $S \in G$, we say that $f \in F$ is the least concept in F containing S, if f is the least concept in F that is consistent with $S \times \{1\} = \{(x, 1) | x \in S\}$. Using this notion, we define the operator $M : G \to F$, such that $M(S)$ is the least concept in F containing S. $M(S)$ is undefined if no concept in F contains S.

We now show an interesting property of the minimally consistent classes. Let F be minimally consistent and let $f \in F$. Let $S \subseteq \Sigma^*$ be a finite set of minimum cardinality such that $f = M(S)$. Then F shatters S. This property is proved in Proposition 7 and can be quite useful. For instance, it is used in Natarajan [15] to prove Lemma 1 for minimally consistent classes.

We need the following supporting results.

PROPOSITION 4. *M is idempotent, i.e., for finite S, $M(M(S)) = M(S)$.*

Proof. Let $f = M(S)$. Surely f is the least concept in F containing f. Hence, $f = M(f) = M(M(S))$. □

PROPOSITION 5. *Let F be minimally consistent, $f \in F$, and let S and T be two finite sets such that $S \subseteq T \subseteq f$. Then, $M(S) \subseteq M(T)$.*

Proof. Since F is minimally consistent and $S \subseteq f$ and $T \subseteq f$, $M(S)$ and $M(T)$ are both defined. Let $S^+ = S \times \{1\}$ and $T^+ = T \times \{1\}$. $M(T)$ is consistent with T^+ and since $S^+ \subseteq T^+$, $M(T)$ is consistent with S^+. But then, $M(S)$ is the least concept in F consistent with S^+. Hence, $M(S) \subseteq M(T)$. □

PROPOSITION 6. *Let F be minimally consistent and let S and T be finite sets such that $M(S)$ and $M(T)$ are both defined. Then $M(S \cup T) = M(M(S) \cup M(T))$.*

Proof. Now $S \subseteq M(S)$ and $T \subseteq M(T)$ and hence $S \cup T \subseteq M(S) \cup M(T)$. Applying M to both sides and invoking Proposition 5, we get $M(S \cup T) \subseteq M(M(S) \cup M(T))$.

Now $S \subseteq S \cup T$. Applying M to both sides and invoking Proposition 5 we get $M(S) \subseteq M(S \cup T)$. Similarly, $M(T) \subseteq M(S \cup T)$. Hence $M(S) \cup M(T) \subseteq M(S \cup T)$. Applying M to both sides and invoking Proposition 5, we get $M(M(S) \cup M(T)) \subseteq M(M(S \cup T))$. By Proposition 4, $M(M(S \cup T)) = M(S \cup T)$ and hence

$$M(M(S) \cup M(T)) \subseteq M(S \cup T).$$

Hence the proposition. □

PROPOSITION 7. *Let F be minimally consistent and let $f \in F$. Let $S \subseteq \Sigma^*$ be a finite set of minimum cardinality such that $f = M(S)$. Then, F shatters S.*

Proof. We show that for every $S_1 \subseteq S$, $M(S_1) \cap S = S_1$. This would immediately imply that F shatters S. Suppose not, i.e., for some $S_1 \subseteq S$, $M(S_1) \cap S = S_2 \neq S_1$. Now, $S = S_2 \cup (S - S_2)$. Apply M to both sides and invoke Proposition 6 to write:

$$M(S) = M(M(S_2) \cup M(S - S_2)).$$

Now, $S_2 = (M(S_1) \cap S) \subseteq M(S_1)$. Applying M to both sides, we get $M(S_2) \subseteq M(M(S_1))$. By the idempotence of M, we can write the above as $M(S_2) \subseteq M(S_1)$. Thus

$$M(S_2) \cup M(S - S_2) \subseteq M(S_1) \cup M(S - S_2).$$

Invoking Proposition 5, we get

$$M(M(S_2) \cup M(S - S_2)) \subseteq M(M(S_1) \cup M(S - S_2)).$$

Substituting this into the equation for $M(S)$, we get

$$M(S) \subseteq M(M(S_1) \cup M(S - S_2)).$$

Invoking Proposition 6, we get

$$M(S) \subseteq M(S_1 \cup (S - S_2)).$$

But $S_1 \cup (S - S_2) \subseteq S$ and by Proposition 5, $M(S_1 \cup (S - S_2)) \subseteq M(S)$. Thus, we have shown that $M(S_1 \cup (S - S_2)) = M(S)$. But then, $S_1 \subseteq (M(S_1) \cap S) = S_2$, and hence $S_1 \subseteq S_2$. By assumption, $S_1 \neq S_2$. Hence, $|S_1 \cup (S - S_2)| < |S|$, implying that S is not of minimum cardinality. This is a contradiction—hence, the proposition. □

3. Time complexity of concept learning. Thus far, we have concerned ourselves with the information complexity of learning, i.e., the number of examples required to learn. In this section, we consider the time-complexity of learning, i.e., the time required to process the examples.

In the previous section, we used the term "algorithm" in a very broad sense, subject only to the statistical restriction of admissibility. This was because we were interested only in the sample complexity of learning, and desired our results to hold over the broadest possible definition. In this section, we use the term "algorithm" in the traditional sense, to mean a finitely representable program with respect to some universal computing machine such as the Turing machine. Of course, this is in addition to the restriction of admissibility.

In order to permit interesting measures of time-complexity, we must specify the manner in which the learning algorithm identifies its approximation to the target concept. In particular, we will require the learning algorithm to identify its output concept using some predetermined naming scheme. To this end, we define the notion of a representation for a class of concepts. In essence, a representation for a class F is an assignment of names for each f in F. The names are simply strings in Σ^*, and each concept in F may have more than one name, as long as two distinct concepts do not share a name.

DEFINITION 13. For a class of concepts F, a representation is a function $I: F \to 2^{\Sigma^*}$. For each $f \in F$, $I(f)$ is the set of *names* for f. I must be such that

(a) each $f \in F$ has at least one name, i.e., for all $f \in F$, $I(f) \neq \emptyset$, and

(b) no two concepts share a name, i.e., for any two distinct f and g in F, $I(f) \cap I(g) = \emptyset$.

The length of a name $i \in I(f)$ is simply the string length $|i|$ of i. Thus, the shortest name for f is the shortest string in $I(f)$. We denote the length of the shortest name for f by $s_{\min}(f)$, relying on the context to specify F and I.

As an aside, we note that if each concept in F were to have a name of finite length, F would be at most countably infinite.

Example 5. As an example, suppose that F were the set of regular sets. One possible representation for F is the mapping I from the regular sets to equivalent regular expressions. (Actually, we should consider the binary encodings of the regular expressions.) For instance, consider the set f of all strings that end in "1." It is well known that f is regular. In fact, the regular expression $r_1 = (0+1)^*1$ generates f, and hence is a name for it in I, i.e., $r_1 \in I(f)$. Similarly, $r_2 = (0+1)^*1^*0^*1$ also generates f and $r_2 \in I(f)$.

Another possible representation for the regular sets is the mapping from the regular sets to the corresponding finite automata. □

We can now specify the form of the output of a learning algorithm. Specifically, we will say that a learning algorithm A learns F in representation I, if A identifies its output concept using names in I. Formally, we have the following definition.

DEFINITION 14. Algorithm A is a probably approximate learning algorithm for *F in representation I*, if

(1) A takes as input ε, $\delta \in (0, 1]$, $n \in \mathbf{N}$.

(2) A may call EXAMPLE, which returns examples for some $f \in F$. The examples are chosen randomly according to an arbitrary and unknown probability distribution P on $\Sigma^{\leqq n}$.

(3) For all concepts $f \in F$ and all probability distributions P on $\Sigma^{\leqq n}$, A outputs $i \in I(g)$ for some $g \in F$, so that with probability at least $(1-\delta)$, $P(f \Delta g) \leqq \varepsilon$.

The aim of this section is to examine the time complexity of learning, and we must construct a measure of the computational time expended by a learning algorithm. In what follows, we will assume that each call of EXAMPLE costs unit time. Surely, the time expended by the learning algorithm will depend on the length of its output. To account for this dependence, we will include the length of the shortest name of the target concept as a parameter in the time complexity measure.

DEFINITION 15. Let A be a learning algorithm for F in representation I. The *time complexity* of A is the function $t: \mathbf{R} \times \mathbf{R} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ such that $t(\varepsilon, \delta, n, s)$ is the maximum number of computational steps consumed by A, the maximum being taken over all runs of A in which the inputs are ε, δ, and n, and the target concept f is such that $s_{\min}(f) \leqq s$.

Thus, the time complexity of A is the time required by A as a function of ε, δ, $s_{\min}(f)$, and n. If this function is bounded by a polynomial in these parameters, we consider the algorithm to be efficient. With this in mind, we give the following definition.

DEFINITION 16. F is *polynomial-time learnable in representation I* if there exists a deterministic learning algorithm A for F with time complexity $O(p(1/\varepsilon, 1/\delta, s_{\min}(f), n))$ where p is a polynomial function of its arguments.

An important difference between the notion of polynomial-sample learnability and that of polynomial-time learnability is that the latter permits the number of examples drawn by the learning algorithm to vary with $s_{\min}(f)$, the length of the output.

Thus, polynomial-time learnability does not imply polymomial-sample learnability. An example of a class that is polynomial-time learnable but not polynomial-sample learnable appears later in this section.

We are interested in identifying the class of pairs F, I, such that F is polynomial-time learnable in I. To this end, we need the following definitions.

DEFINITION 17. Algorithm Q is said to be a *fitting* for a class F in index I if

(1) Q takes as input a set of examples $S \subseteq \Sigma^* \times \{0, 1\}$.

(2) If there exists $f \in F$ such that f is consistent with S, Q outputs $i \in I(g)$, such that $g \in F$ and g is consistent with S.

In the above, we say that g is the concept identified by Q in its output. Abusing notation, we extend the notation $s_{\min}(\ )$ to sets of examples as follows. For a set of examples S, $s_{\min}(S)$ is the length of the shortest name of any concept in F consistent with S, i.e., $s_{\min}(S) = \min\{s_{\min}(f) | f \in F, S \subseteq graph(f)\}$. If no concept in F is consistent with S, $s_{\min}(S)$ is infinity.

DEFINITION 18. Let Q be a deterministic program that is a fitting for F in I. If, on input S, Q runs in time polynomial in the length of its input and $s_{\min}(S)$, we say that Q is a *polynomial-time fitting*.

With these definitions in hand, we can state the following theorem.

THEOREM 3. *Let F be a class of concepts of polynomial dimension and let I be a representation for F. F is polynomial-time learnable in I if there exists a polynomial-time fitting for F in I.*

Proof. Let Q be a polynomial-time fitting for F. The following is a polynomial-time learning algorithm for F in I.

Learning Algorithm A_3
input: ε, δ, n.
begin:

Call EXAMPLE $\frac{1}{\varepsilon}\left((n+2)\dim_{vc}(F_n)\ln(2)+\ln\left(\frac{1}{\delta}\right)\right)$ times.

let S be the set of examples seen.
output $Q(S)$

end

We can prove that A_3 is a probably approximate learning algorithm for F using the methods of the proof of Theorem 1. It remains to bound the running time of the algorithm. Since each call of EXAMPLE costs unit time, A runs in time polynomial in $1/\varepsilon$, $1/\delta$, and n, except for the time taken to simulate Q. Now Q runs in time polynomial in the size of its input and in $s_{\min}(S)$. By definition, $s_{\min}(S) \leqq s_{\min}(f)$. The length of the description of S is the size of Q's input. Surely, this is at most $|S|n$ and hence is polynomial in n, $1/\varepsilon$, and $1/\delta$. It follows that A runs in time polynomial in $1/\varepsilon$, $1/\delta$, n, and $s_{\min}(f)$. □

Example 6. Consider the class F of axis-parallel rectangles on $\mathbf{N} \times \mathbf{N}$ again. As a representation for the class, we will use the binary encoding of the lower left and upper right vertices of each rectangle. We now show that F has a polynomial-time fitting in the representation chosen.

Fitting Q
input S: set of examples.
begin

compute $x_{\max}$ and $x_{\min}$, the maximum and minimum
values of the x-coordinates of the positive examples of S.
compute $y_{\max}$ and $y_{\min}$, the maximum and minimum

values of the y-coordinates of the positive examples of S.
output $((x_{\min}, y_{\min}), (x_{\max}, y_{\max}))$
end

In Example 2, we showed that F was of polynomial dimension. From Theorem 3, it follows that F is polynomial-time learnable in the representation of diagonal points. □

Theorem 3 holds only when F is of polynomial dimension. When F is not of polynomial dimension, sufficient conditions for polynomial-time learnability can be established in terms of Occam fittings. See [6], [7]. For example, the class of concepts comprised of finitely many intervals on **N** is not of polynomial dimension. However, it is polynomial-time learnable. See [7].

We now prove a weak form of the converse of Theorem 3. We need the following definitions.

DEFINITION 19. A coin-tossing algorithm Q is a *randomized fitting* for F in I if

(1) Q takes as input a set of examples $S \subseteq \Sigma^* \times \{0, 1\}$.

(2) If there exists $f \in F$ such that f is consistent with S, then with probability greater than $\frac{1}{2}$, Q outputs $i \in I(g)$, such that $g \in F$ and g is consistent with S. If Q fails to output such a name i, Q outputs nothing.

If Q runs in time polynomial in the length of its input and in $s_{\min}(S)$, we say it is a *random-polynomial-time fitting.*

The following definition concerns the complexity of testing for membership in a concept $f \in F$, given a name $i \in I(f)$.

DEFINITION 20. I is *polynomial-time verifiable* if there exists a deterministic algorithm V such that

(1) V takes as input strings $i, x, y \in \Sigma^*$.

(2) If i is such that $i \in I(f)$ for some $f \in F$, V accepts its input if and only if $y = f(x)$.

(3) V runs in time polynomial in the length of its input.

THEOREM 4. *Let F be a class of concepts and let I be a polynomial-time verifiable representation for F. F is polynomial-time learnable in I only if F has a random-polynomial-time fitting in I.*

Proof. Assume that F is polynomial-time learnable in representation I by an algorithm A and that I is polynomial-time verifiable by a program V. We need to show that there exists a randomized fitting for F that runs in polynomial time. The following is such a fitting. In words, Q takes as input a set of examples S and then runs the learning algorithm A as follows: at each call of EXAMPLE, Q gives A a randomly chosen example from S. Since A is a probably approximate learning algorithm, the name i output by A is likely to be that of a concept g that is consistent with S. Using the verifier V, Q checks to see if g is indeed consistent with S. If so, Q outputs i.

Fitting Q
input: $S \subseteq \Sigma^* \times \{0, 1\}$.
begin
let $\varepsilon = 1/(|S|+1)$ and $\delta = \frac{1}{4}$.
let n be the length of the longest example in S.
run A on inputs ε, δ, and n.
on each call of EXAMPLE by A
give A a randomly chosen element of S.
let i be the representation output by A.
if for each $(x, y) \in S$, V accepts (i, x, y) **then** output i
else output nothing.
end

In essence, the fitting Q runs A with S as examples. If $\varepsilon < 1/|S|$ and $\delta = \frac{1}{4}$, the name output by A must be that of a concept consistent with S with probability at least $(1-\frac{1}{4}) = \frac{3}{4}$. By invoking the verifier V, Q checks to see if A's output is consistent with S. If so, Q halts successfully. If not, Q outputs nothing. Thus, with probability $\frac{3}{4}$, Q outputs the name of a concept in F that is consistent with S.

It remains to show that Q runs in polynomial time. Since A is a polynomial-time learning algorithm, A runs in time polynomial in n, $1/\varepsilon$, $1/\delta$, and in $s_{\min}(f)$, where f is the target concept. But, f could be any concept in f consistent with S, and so A must run in time polynomial in $s_{\min}(S)$. Thus, Q runs in time polynomial in the size of its input and in $s_{\min}(S)$, and hence is a random-polynomial-time fitting. □

At this point, it is important to point out that our definition of learnability is perhaps a little too restrictive. We illustrate our point thus: A k-term-DNF formula is a boolean formula with k terms, each term being the conjunction of literals. A k-CNF formula is a conjunction of clauses, where each clause is a disjunction of at most k literals. We can view strings on Σ^* as assignment vectors to the variables in a boolean formula, and hence the set of satisfying assignments of the formula is a concept on Σ^*. Let us call the class of concepts representable by k-term-DNF formulae k-term-DNF concepts, and similarly, we have the k-CNF concepts. Clearly, the k-term-DNF formulae form a representation for the k-term-DNF concepts. Pitt and Valiant [19] show that the k-term-DNF concepts are not polynomial-time learnable in the representation of k-term-DNF formulae, unless $NP = RP$. Note that the dimension of the k-term-DNF concepts is polynomial, and hence the class is polynomial-sample learnable, i.e., it is possible to infer a good approximation to the target concept from a small number of examples. Yet, finding a k-term-DNF formula to represent the approximation is computationally difficult. Suppose we allow the learning algorithm to output a k-CNF formula as an approximation to the k-term-DNF target concept. That is, the learning algorithm outputs a k-CNF formula which may not be equivalent to any k-term-DNF formula, but is guaranteed to be a good approximation to the target concept. [19] shows that such a learning algorithm can learn the k-term-DNF concepts in polynomial time. In the more general setting, we can permit the learning algorithm to output any program that is a good approximation to the target concept, under the restriction that the run time of the output program be bounded by a predetermined polynomial. This generalized notion of learning is called "prediction" and is discussed in Pitt and Warmuth [21].

Example 7. Recall the class F of regular sets with the representation I of finite automata. [20] shows that if $P \neq NP$, there does not exist a polynomial-time fitting for F in I. Assuming the existence of trapdoor functions, Kearns and Valiant [12] show that the regular sets are not predictable. □

Analogous to Theorems 3 and 4, we can state results for learning with one-sided error. Since these results are straightforward analogs of Theorems 3 and 4, we leave their details to the reader.

4. Learning functions on discrete domains. In the foregoing, we were concerned with learning approximations to concepts on Σ^*. In the more general setting, one may consider learning functions from Σ^* to Σ^*. To do so, we must first modify our definitions suitably and generalize our formulation of the problem.

We consider classes of total functions from Σ^* to Σ^*. An example for a function f is a pair (x, y) such that $y = f(x)$. A learning algorithm for a class of functions is an algorithm that attempts to infer an approximation to a function in F from examples for it. The learning algorithm has at its disposal a subroutine EXAMPLE, which at each call produces a randomly chosen example for the function to be learned. The

function f for which examples are provided is the *target function.* The examples are chosen according to an arbitrary and unknown probability distribution P in that the probability that a particular example $(x, f(x))$ will be produced at any call is $P(x)$.

As in the case of concepts, we define the notion of a probably approximate learning algorithm.

Algorithm A is a *probably approximate* learning algorithm for F if

(1) A takes as input reals ε, $\delta \in (0, 1]$, and $n, r \in \mathbf{N}$.

(2) A may call EXAMPLE, which returns examples for some $f \in F$, where f is such that for all $x \in \Sigma^{\leqq n}$, $f(x) \in \Sigma^{\leqq r}$. The examples are chosen randomly and independently according to an arbitrary and unknown probability distribution P on $\Sigma^{\leqq n}$.

(3) For all $f \in F$ (such that for every $x \in \Sigma^{\leqq n}$, $f(x) \in \Sigma^{\leqq r}$), and all probability distributions P on $\Sigma^{\leqq n}$, with probability at least $(1-\delta)$, A outputs a function $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

Notation. For two functions f and g, $f \Delta g = \{x | f(x) \neq g(x)\}$.

DEFINITION 21. The *sample complexity* of a learning algorithm A is the function $s: \mathbf{R} \times \mathbf{R} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ such that $s(\varepsilon, \delta, n, r)$ is the maximum number of calls of EXAMPLE by A, the maximum being taken over all runs of A on input ε, δ, n, and r.

DEFINITION 22. A class of functions is said to be polynomial-sample learnable if there exists a learning algorithm for F with sample complexity $O(p(1/\varepsilon, 1/\delta, n, r))$, where p is a polynomial function of its arguments.

We can now generalize the notion of shattering as follows.

DEFINITION 23. Let F be a class of functions from a set X to Y. We say F *shatters* a set $S \subseteq X$ if there exist two functions $f, g \in F$ such that

(1) for all $x \in S, f(x) \neq g(x)$;

(2) for all $S_1 \subseteq S$, there exists $e \in F$ such that e agrees with f on S_1 and with g on $S - S_1$, i.e., for all $x \in S_1$, $e(x) = f(x)$; for all $x \in S - S_1$, $e(x) = g(x)$.

DEFINITION 24. Let F be a class of functions from a set X to a set Y. The *generalized dimension* of F, denoted by $\dim_g(F)$, is the least integer d such that every set shattered by F is of cardinality at most d.

In the interest of simplicity, we drop the prefix "generalized" unless it is required by the context. Note that if Y is of cardinality two, then F could well be the indicator functions of a class of concepts on X. In this case, the generalized dimension is the same as the Vapnik-Chervonenkis dimension.

We now define an asymptotic variant of the generalized dimension.

DEFINITION 25. Let $f: \Sigma^* \rightarrow \Sigma^*$. For $n, r \in \mathbf{N}$, the *projection* $f_{n,r}$ of f on $\Sigma^{\leqq n} \times \Sigma^{\leqq r}$ is

(1) Undefined if there exists $x \in \Sigma^{\leqq n}$ such that $f(x) \notin \Sigma^{\leqq r}$.

(2) Else, it is the function $f_{n,r}: \Sigma^{\leqq n} \rightarrow \Sigma^{\leqq r}$ such that for all $x \in \Sigma^{\leqq n}$, $f_{n,r}(x) = f(x)$.

The projection of a class of functions F is the class $F_{n,r}$ given by $F_{n,r} = \{f_{n,r} | f \in F, f_{n,r}$ is defined$\}$.

DEFINITION 26. The *asymptotic dimension* of a class of functions F is the function $d: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ such that for all $n, r, \dim_g(F_{n,r}) = d(n, r)$. We denote the asymptotic dimension of F by $\dim(F)$.

For simplicity, we drop the prefix asymptotic, relying on the context for clarity. If $\dim(F)$ is $O(p(n, r))$ for some polynomial p, we say that F is of polynomial dimension. We can now generalize our shattering lemma for functions as follows. The lemma is a generalization of Lemma 1 and is proved using similar techniques.

LEMMA 2 (Generalized Shattering Lemma). *Let $F_{n,r}$ be of dimension d. Then,* $2^d \leqq |F_{n,r}| \leqq 2^{d(n+2r+3)}$.

Proof. The first inequality is immediate from the definition of shattering.

The second is proved through the following claim.

CLAIM 2. *Let X and Y be two finite sets and let H be a set of total functions from X to Y. If $d = \dim_g(H)$ then,*

$$|H| \leqq |X|^d |Y|^{2d}.$$

Proof. This is proved by induction on $|X|$.

Basis. This is clearly true for $|X| = 1$, for all $|Y|$. Also, we can assume $d \geqq 1$ as the hypothesis holds trivially otherwise.

INDUCTION. Assume true for $|X| = k$ and $|Y| = l$, and prove true for $|X| = k+1$ and $Y = l$. Let $X = \{x_1, x_2, \cdots, x_{k+1}\}$ and $y = \{y_1, y_2, \cdots, y_l\}$. Define the sets of functions $H_i \subseteq H$ as follows.

$$H_i = \{f \mid f \in H, f(x_1) = y_i\}.$$

Also, define the sets of functions H_{ij} and H_0 as follows:

$$\text{for } i \neq j, \qquad H_{ij} = \{f \mid f \in H_i, \text{ there exists } g \in H_j \text{ such that } f = g \text{ on } X - \{x_1\}\}.$$

$$H_0 = H - \bigcup_{i \neq j} H_{ij}.$$

Now,

$$|H| = |H_0| + \left| \bigcup_{i \neq j} H_{ij} \right| \leqq |H_0| + \sum_{i \neq j} |H_{ij}|.$$

We seek bounds on the quantities on the right-hand side of the last inequality. By definition, the functions in H_0 are all distinct on the k elements of $X - \{x_1\}$. Furthermore, the largest set shattered in H_0 must be of cardinality no greater than d. By the inductive hypothesis, we have

$$|H_0| \leqq k^d l^{2d}.$$

And then, every H_{ij} shatters a set S of cardinality at most $d-1$. Otherwise, H would shatter $S \cup \{x_1\}$, which would be of cardinality greater than d. Also, since the functions in H_{ij} are all distinct on $X - \{x_1\}$, we have by the inductive hypothesis,

$$\text{for } i \neq j, \qquad |H_{ij}| \leqq k^{d-1} l^{2(d-1)}.$$

Combining the last three inequalities, we have

$$\begin{aligned} |H| &\leqq k^d l^{2d} + \sum_{i \neq j} k^{d-1} l^{2(d-1)} \\ &\leqq k^d l^{2d} + l^2 k^{d-1} l^{2(d-1)} \\ &\leqq k^d l^{2d} + k^{d-1} l^{2d} \leqq (k+1)^d l^{2d}, \end{aligned}$$

which completes the proof of the claim.

Returning to the lemma, we have $X = \Sigma^{\leqq n}$, $Y = \Sigma^{\leqq r}$ and hence $k \leqq 2^{n+1}$ and $l \leqq 2^{r+1}$. Thus, $|F_{n,r}| \leqq 2^{d(n+2r+3)}$. □

We can now state the main theorem of this section.

THEOREM 5. *A class of functions is polynomial-sample learnable if and only if it is of polynomial dimension.*

Proof. This proof is similar to the proof of Theorem 1, except that we need to use the generalized notion of shattering and the corresponding generalized shattering lemma. □

4.1. Time complexity of function learning. Analogous to our development of time-complexity considerations for concept learning, we can obtain corresponding results for learning functions. To this end, we present the following analogs of the definitions given earlier in the context of concept learning.

The notion of a representation for a class of functions is identical to that for a class of concepts. Also, the notion of a graph set, and the definition of $s_{\min}$ carry over without changes.

DEFINITION 27. Algorithm A is a probably approximate learning algorithm for *F in representation I*, if

(1) A takes as input $\varepsilon, \delta \in (0, 1]$, $n, r \in \mathbf{N}$.

(2) A may call EXAMPLE, which returns examples for some $f \in F$, where f is such that, for all $x \in \Sigma^{\leqq n}$, $f(x) \in \Sigma^{\leqq r}$. The examples are chosen randomly according to an arbitrary and unknown probability distribution P on $\Sigma^{\leqq n}$.

(3) For all $f \in F$, (such that for every $x \in \Sigma^{\leqq n}$, $f(x) \in \Sigma^{\leqq r}$), and all probability distributions P on $\Sigma^{\leqq n}$, A outputs $i \in I(g)$ for some $g \in F$, so that with probability at least $(1-\delta)$, $P(f \Delta g) \leqq \varepsilon$.

DEFINITION 28. Let A be a learning algorithm for a class of functions F. The *time complexity* of A is the function $t: \mathbf{R} \times \mathbf{R} \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ such that $t(\varepsilon, \delta, s_{\min}(f), n, r)$ is the maximum number of computational timesteps consumed by A, the maximum being taken over all runs of A in which the inputs are ε, δ, n, and r, and the target function f is such that $s_{\min}(f) \leqq s$.

DEFINITION 29. F is *polynomial-time learnable in representation I* if there exists a deterministic learning algorithm A for F with time complexity $O(p(1/\varepsilon, 1/\delta, s_{\min}(f), n, r))$, where p is a polynomial function of its arguments.

We are interested in identifying the class of pairs F, I, such that F is polynomial-time learnable in I. To this end, we carry over the definitions of a fitting, a random-polynomial-time fitting, and polynomial-time verifiability to this setting. These definitions require no changes and are not repeated below.

A class of functions from Σ^* to Σ^* is said to be of *polynomial expansion* if there exists a polynomial p such that for all $x \in \Sigma^*$, $|f(x)| \leqq p(|x|)$.

With these definitions in hand, we can state the following theorem.

THEOREM 6. *Let F be a class of functions and let I be a representation for F.*

(1) *F is polynomial-time learnable in I if* (a) *F is of polynomial dimension,* (b) *F is of polynomial expansion, and* (c) *there exists a polynomial-time fitting for F in I.*

(2) *If F is polynomial-time learnable in I, and I is polynomial-time verifiable, there exists a random-polynomial-time fitting for F in I.*

Proof. This proof is similar to the proofs of Theorems 3 and 4. □

Example 8. Let p be a permutation on k items. The function $f^{(p)}$ from Σ^* to Σ^* is defined as follows. Let $x \in \Sigma^*$. If x is not of length k, $f^{(p)}(x) = x$. Else, $f^{(p)}(x) = y$, where y is the string obtained by applying the permutation p to the k bits of x. We call $f^{(p)}$ a permutation function. Let F be the set of all permutation functions, for all values of k.

As representation I for F, we use the following: Let $f^{(p)} \in F$, where p is a permutation on k items. A name for f in I is a binary encoding of the map of p. Specifically, the binary encoding of the bipartite graph $G = (U, V, E)$, where the vertex sets U and V each have k vertices and the edge set E has an edge from vertex i in U to vertex $j \in V$, if p maps item i to item j.

We claim that F is of polynomial dimension. Note that for $r \geqq n$, $F_{n,r} = F_{n,n}$. Thus,

$$|F_{n,r}| \leqq |F_{n,n}| \leqq \sum_{k \leqq n} k! \leqq n^{n+1}.$$

By Lemma 2, $\dim_g(F_{n,r}) \leqq (n+1) \log n$, which is polynomial in n and r.

Next, we show that F has a polynomial-time fitting in I. Suppose we are given a set of examples S and are required to find a permutation function consistent with the

examples. If, for all $(x, y) \in S$, $x = y$, we can simply output the identity permutation for some value of k. Otherwise, there must exist a value of k such that for every $(x, y) \in S$ for which $x \neq y$, $|x| = k$. Delete all $(x, y) \in S$ for which $|x| \neq k$. Construct a bipartite graph $G = (U, V, E)$, where the vertex sets U and V each have k vertices and the edge set E has an edge between every pair of vertices (u, v), $u \in U$, and $v \in V$. Edges in G are denoted (i, j), $i \in U$, and $j \in V$.

We now delete each edge (i, j) from G, such that S contains an example (x, y) where the ith bit of x and the jth bit of y do not agree. Note that any permutation that maps i to j cannot be consistent with such an example (x, y). This deletion process can be carried in $O(|S|k^2)$ time.

It is easy to see that any perfect matching of the resulting graph G is the map of a permutation that is consistent with S. Since perfect matchings of bipartite graphs can be computed in time polynomial in the size of the graph (Tarjan [23]), we can conclude that F has a polynomial-time fitting.

By Theorem 6, F is polynomial-time learnable. □

5. Learning functions on continuous domains. Thus far, we have explored the learnability of classes of sets and functions defined on discrete domains, such as the strings of a finite alphabet. This permitted us to inquire into the number of examples needed for learning as an asymptotic function of the length of the strings over which the probability distribution was significant. We now consider the learnability of classes defined on continuous domains such as the reals. Here, we are interested in identifying classes of concepts and functions that are learnable from finitely many examples, a notion that we call "learnability," as opposed to the notion of polynomial-sample learnability.

For the case of classes of concepts, Blumer et al. [8] present conditions necessary and sufficient for learnability. Their results use powerful tools in classical probability theory developed in [25]. In the following, we review these results briefly and then go on to present learnability results for classes of functions, relying in part on the results of [8].

As is to be expected, a concept on $\mathbf{R}$ is a subset of $\mathbf{R}$, and a class of concepts in $\mathbf{R}$ is a subset of $2^{\mathbf{R}}$. Similarly, a class of functions on $\mathbf{R}$ is a set of functions from $\mathbf{R}$ to $\mathbf{R}$.

DEFINITION 30. Let F be a class of concepts on $\mathbf{R}$. We say F is *learnable* if there exists an algorithm A such that

(1) A takes as input $\varepsilon, \delta \in (0, 1]$.

(2) A may call EXAMPLE. EXAMPLE returns examples for some concept f in F, where the examples are chosen randomly according to an arbitrary and unknown probability distribution P on $\mathbf{R}$. The number of calls of EXAMPLE must be finite, although it may depend on ε, δ, and P.

(3) For all probability distributions P and all f in F, with probability $(1-\delta)$, A outputs $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

Notation. Here, for $S \subseteq \mathbf{R}$, $P(S) = \int_S dP$.

The following theorem is adapted from [8]. Actually, the authors obtain this theorem for a slightly different definition of learnability. Their definition of learnability can be shown to be equivalent to ours (see Haussler et al. [9]).

THEOREM 7 [8]. *A class of concepts F on $\mathbf{R}$ is learnable if and only if* $\dim_{vc}(F)$ *is finite.*

There are some measurability assumptions necessary for Theorem 7. See [6], [8].

We now formalize the notion of learnability of classes of functions on the reals.

DEFINITION 31. Let F be a class of functions from $\mathbf{R}$ to $\mathbf{R}$. We say F is *learnable* if there exists an algorithm A such that

(1) A takes as input ε, $\delta \in (0, 1]$.

(2) A may call EXAMPLE. EXAMPLE returns examples for some function f in F, where the examples are chosen randomly according to an arbitrary and unknown probability distribution P on $\mathbf{R}$. The number of calls of EXAMPLE must be finite, although it may depend on ε, δ, and P.

(3) For all probability distributions P and all functions f in F, with probability $(1-\delta)$, A outputs $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

We now state the main theorem of this section. The theorem is not tight in the sense that the necessary and sufficient conditions do not match. (In [16], a tight version of the theorem was incorrectly reported.) Indeed, we will identify a learnable class of functions that sits in the gap between these conditions.

THEOREM 8. *A class of functions F from $\mathbf{R}$ to $\mathbf{R}$ is learnable*

(1) *if* $\dim_{vc}$ *(graph(F)) is finite*;

(2) *only if* $\dim_g$ *(F) is finite.*

Proof. The "if" direction of the proof follows from Theorem 7. Essentially, the "if" condition implies that the class *graph*(F) is learnable, whence it follows that the class F is learnable.

The "only if" direction of the proof is identical to that of Theorem 4, which in turn follows the arguments of Theorem 1. □

While Theorem 8 is not tight, it appears that tightening it is a rather difficult task. We conjecture that the "if" condition should match the "only if" condition. To give the reader a flavor of the difficulties involved in tightening Theorem 8, we give an example of a class of functions F that lies in the gap between the necessary and sufficient conditions of Theorem 8. Specifically,

(1) $\dim_g (F)$ is finite,

(2) $\dim_{vc}$ (*graph*(F)) is infinite,

(3) F is learnable.

Example 9. For $\alpha \in \mathbf{N}$, the ith bit of alpha is the ith bit in the binary representation of α, counting from the right. For instance, the second bit of 13 is 0 since the binary representation of 13 is 1101. Define the function $f^{(\alpha)}: \mathbf{R} \rightarrow \mathbf{R}$ as follows.

$$f^{(\alpha)}(x) = \begin{cases} \alpha & \text{if } x \in \mathbf{N} \text{ and the } x\text{th bit of } \alpha \text{ is } 1, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $f^{(0)}$ is zero everywhere, and $f^{(13)}(x) = 13$ for $x = 1, 3, 4$ and zero elsewhere.

Define the class F as follows.

$$F = \{f^{(\alpha)} \mid \alpha \in \mathbf{N}\}.$$

CLAIM 3. $\dim_g (F) = 1$.

Proof. Suppose F shatters a set of size greater than one. Then F must shatter a set of size 2. Let $S = \{a, b\}$ be such a set. It follows from the definition of F that $a, b \in \mathbf{N}$. By the definition of shattering, there exist three functions $f^{(\alpha)}$, $f^{(\beta)}$, and $f^{(\gamma)}$ in F such that $f^{(\alpha)}(a) \neq f^{(\beta)}(a)$, $f^{(\alpha)}(b) \neq f^{(\beta)}(b)$, and $f^{(\gamma)}(a) = f^{(\alpha)}(a)$, $f^{(\gamma)}(b) = f^{(\beta)}(b)$. Since $f^{(\alpha)}(a) \neq f^{(\beta)}(a)$, at least one of them must be nonzero. Without loss of generality, assume that $f^{(\alpha)}(a)$ is nonzero. Now, $f^{(\alpha)}(a) \neq 0$ implies that $f^{(\alpha)}(a) = \alpha$. Since $f^{(\gamma)}(a) = f^{(\alpha)}(a)$, $f^{(\gamma)}(a) = \alpha$. But then, $f^{(\gamma)}(a) \neq 0$ implies that $f^{(\gamma)}(a) = \gamma$, and hence $\alpha = \gamma$. This contradicts the assumption that $f^{(\gamma)}(b) = f^{(\beta)}(b) \neq f^{(\alpha)}(b)$—hence, the claim. □

CLAIM 4. $\dim_{vc}$ *(graph(F)) is infinite.*

Proof. Let S_1 be any arbitrarily large but finite subset of **N**. Consider $S = S_1 \times \{0\}$. Now, *graph*(F) shatters S, since for any subset S_2 of S, there exists a set $f \in F$ such that *graph*$(f) \cap S = S_2$. To see this, let S_3 be such that $S - S_2 = S_3 \times \{0\}$. We can pick an integer $\alpha \in \mathbf{N}$, such that for all $x \in S_3$, the xth bit of α is 1, with all other bits in α being zero. Thus $f^{(\alpha)}(x) \neq 0$ if and only if $x \in S_3$—hence, the claim. □

CLAIM 5. *F is learnable.*

Proof. The following is a learning algorithm for F.

Learning Algorithm A_4.
input $\varepsilon, \delta \in (0, 1]$.
begin

call EXAMPLE $\left(\frac{1}{\varepsilon}\right) \ln \left(\frac{1}{\delta}\right)$ times.

if any of the examples seen is of the
form (x, y), $y \neq 0$,
then output $f^{(y)}$.
else output $f^{(0)}$.

end

We now show that the probabilities work out for the above algorithm. Suppose the function to be learned were $f^{(\alpha)}$, for some $\alpha \neq 0$. Then, if

$$P(f^{(\alpha)} \Delta f^{(0)}) \geqq \varepsilon,$$

with probability $(1 - \delta)$, in $(1/\varepsilon) \ln (1/\delta)$ examples there must be an example of the form (x, α). In this case, the algorithm will output $f^{(\alpha)}$, implying that, with probability $(1 - \delta)$, the algorithm learns the unknown function exactly—hence, the claim. □

In the foregoing, we considered functions on the reals, requiring that on a randomly chosen point, with high probability, the learner's approximation agrees exactly with the target function. This requires infinite precision arithmetic and hence is largely of technical interest. If all the computations are carried out only to some finite precision, Theorem 5 would apply directly. Alternatively, we could require that the learned function approximate the target function with respect to some norm. This is discussed in [17], [18], and [10].

6. Appendix. We show that eliminating the length parameter n as input to the learning algorithm leaves the results of the paper invariant.

Let us call the learning paradigm of § 2 Framework 1. We define Framework 2 as follows.

DEFINITION 32. Algorithm A is a *probably approximate* learning algorithm for F if:

(1) A takes as input reals $\varepsilon, \delta \in (0, 1]$.

(2) A may call EXAMPLE, which returns examples for some $f \in F$. The examples are chosen randomly and independently according to an arbitrary and unknown probability distribution P on Σ^*.

(3) For all concepts $f \in F$ and all probability distributions P on Σ^*, with probability at least $(1 - \delta)$, A outputs a concept $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

DEFINITION 33. The *sample complexity* of a learning algorithm A is the function $s: \mathbf{R} \times \mathbf{R} \times \mathbf{N} \to \mathbf{N}$ such that $s(\varepsilon, \delta, n)$ is the maximum number of calls of EXAMPLE by A, the maximum being taken over all runs of A on input ε and δ, during which n is the length of the longest example seen by A.

Suppose that A is a learning algorithm for a class of concepts F in Framework 2 with sample complexity $s(\varepsilon, \delta, n)$. We show how to construct a learning algorithm $\hat{A}$ for F in Framework 1 with the same sample complexity. On input ε, δ, n, $\hat{A}$ runs A on inputs ε, δ, and outputs A's output. Since the distribution P is on $\Sigma^{\leqq n}$, the length of the longest example seen by A will be n and hence the number of calls of EXAMPLE will not exceed $s(\varepsilon, \delta, n)$. Also, the run time of $\hat{A}$ is but an additive constant over the run time of A.

Conversely, let A be a learning algorithm for F in Framework 1 with sample complexity $s(\varepsilon, \delta, n)$. The following is a learning algorithm $\hat{A}$ for F in Framework 2. In words, $\hat{A}$ first estimates a length n such that, with probability $1-(\delta/3)$, $\sum_{|x|\leqq n} P(x) \geqq 1-(\varepsilon/2)$. Then, $\hat{A}$ runs A with inputs $\varepsilon/2$, $\delta/3$, n. $\hat{A}$ has to make sure that A never receives an example of length greater than n. To achieve this, $\hat{A}$ draws a suitable number of examples so that, with probability $1-(\delta/3)$, a sufficient number of the examples obtained will be of length at most n. $\hat{A}$ then discards the examples longer than n and uses the remainder in its simulation of A. A formal proof that $\hat{A}$ is indeed a learning algorithm for F in Framework 2 is left to the reader.

Learning Algorithm $\hat{A}$
begin

Call EXAMPLE $\left(\frac{2}{\varepsilon}\right) \ln\left(\frac{3}{\delta}\right)$ times.

Let n be the length of the longest example so obtained.

Let $m = s(\varepsilon/2, \delta/3, n)$.

Call EXAMPLE $\frac{m}{\ln(2/\varepsilon)} \ln\left(\frac{3m}{\delta}\right)$ times.

Let S be the sequence of examples so obtained.

Delete from S examples whose length exceed n.

if S has fewer than m examples left **then** halt.

else

run A on input $(\varepsilon/2, \delta/3, n)$.

on the ith call of EXAMPLE by A

give A the ith example from S.

output the output of A.

end

end

Thus, F is polynomial-sample learnable in Framework 2 if and only if it is polynomial-sample learnable in Framework 1. Also, F is polynomial-time learnable in Framework 2 if and only if it is polynomial-time learnable in Framework 1.

Acknowledgments. I thank R. Kannan, P. Tadepalli, and T. Mitchell for many useful discussions. Most of all, many thanks to the referees who helped immensely through careful readings and insightful comments.

REFERENCES

[1] D. ANGLUIN AND C. H. SMITH (1983), *Inductive inference: Theory and methods*, Comput. Surveys, 15, pp. 237-269.

[2] D. ANGLUIN (1986), *Learning regular sets from queries and counter-examples*, Tech. Report TR-464, Department of Computer Science, Yale University, New Haven, CT.

[3] P. ASSOUAD (1983), *Densité et dimension*, Ann. Inst. Fourier, 33, pp. 233-282.

[4] P. BERMAN AND R. ROOS (1987), *Learning one-counter languages in polynomial time*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 61-67.

[5] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. WARMUTH (1986), *Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension*, in Proc. 18th ACM Symposium on Theory of Computing, pp. 273-282.

[6] ——— (1987) *Learnability and the Vapnik-Chervonenkis dimension*, Tech. Report UCSC-CRL-87-20, Department of Computer Science, University of California, Santa Cruz, CA.

[7] ——— (1987), *Occam's Razor*, Inform. Process. Lett., 24, pp. 377-380.

[8] ——— (1989), *Learnability and the Vapnik-Chervonenkis Dimension*, J. Assoc. Comput. Mach., 36, pp. 929-965.

[9] D. HAUSSLER, M. KEARNS, N. LITTLESTONE, AND M. K. WARMUTH (1988), *Equivalence of models for polynomial learnability*, Tech. Report UCSC-CRL-88-06, Department of Computer Science, University of California, Santa Cruz, CA.

[10] D. HAUSSLER (1989), *Generalizing the* PAC *model: Sample size bounds from metric dimension-based uniform convergence results*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, pp. 40-45.

[11] M. KEARNS, M. LI, L. PITT, AND L. G. VALIANT (1987), *On the learnability of Boolean formulae*, in Proc. 19th ACM Symposium on Theory of Computing, pp. 285-295.

[12] M. KEARNS AND L. G. VALIANT (1989), *Cryptographic limitations on learning Boolean formulae and finite automata*, in Proc. 21st ACM Symposium on Theory of Computing, pp. 421-432.

[13] P. LAIRD (1987), *Learning from data good and bad*, Ph.D thesis, Department of Computer Science, Yale University, New Haven, CT.

[14] R. MICHALSKI, T. MITCHELL, AND J. CARBONELL (1983), *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, CA.

[15] B. K. NATARAJAN (1986), *On learning Boolean functions*, Tech. Report TR-86-17, Carnegie-Mellon Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA; also in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 296-304.

[16] B. K. NATARAJAN AND P. TADEPALLI (1988), *Two new frameworks for learning*, in Proc. 5th International Symposium on Machine Learning, American Association of Artificial Intelligence and Association for Computing Machinery, Ann Arbor, MI, 1988, pp. 402-415.

[17] B. K. NATARAJAN (1989), *Some results on learning*, Tech. Report CMU-RI-TR-89-6, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

[18] ——— (1989), *Probably approximate learning over classes of distributions*, submitted.

[19] L. PITT AND L. G. VALIANT (1986), *Computational limitations on learning, from examples*, Tech. Report TR-05-86, Department of Computer Science, Harvard University, Cambridge, MA.

[20] L. PITT AND M. WARMUTH (1989), *The minimum consistent* DFA *problem cannot be approximated within any polynomial*, in Proc. 21st ACM Symposium on Theory of Computing, pp. 421-432.

[21] ——— (1991), *Prediction preserving reducibility*, J. Comput. System Sci., to appear.

[22] R. RIVEST AND R. E. SCHAPIRE (1987), *Diversity based inference of finite automata*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, pp. 78-87.

[23] R. E. TARJAN (1983), *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA.

[24] L. G. VALIANT (1984), *A theory of the learnable*, in Proc. 16th ACM Symposium on Theory of Computing, pp. 436-445.

[25] V. N. VAPNIK AND A. YA. CHERVONENKIS (1971), *On the uniform convergence of relative frequencies*, Theory Probab. Appl., pp. 264-280.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 352-375, April 1991

© 1991 Society for Industrial and Applied Mathematics
011

EFFICIENT PARALLEL ALGORITHMS FOR TESTING k-CONNECTIVITY AND FINDING DISJOINT s-t PATHS IN GRAPHS*

SAMIR KHULLER† AND BARUCH SCHIEBER‡

Abstract. An efficient parallel algorithm for testing whether a graph G is k-vertex connected is presented. The algorithm runs in $O(k^2 \log n)$ time and uses $(n+k^2)kC(n, m)$ processors on a CRCW PRAM, where n and m are the number of vertices and edges of G, and $C(n, m)$ is the number of processors required to compute the connected components of G in logarithmic time. For fixed k, the algorithm runs in logarithmic time and uses $nC(n, m)$ processors. To develop our algorithm, an efficient parallel algorithm is designed for the following *disjoint s-t paths* problem. Given a graph G, and two specified vertices s and t, find k vertex disjoint paths between s and t, if they exist. If no such paths exist, find a set of at most $k-1$ vertices whose removal disconnects s and t. The parallel algorithm for this problem runs in $O(k^2 \log n)$ time and uses $kC(n, m)$ processors. The way to modify the algorithm to find k-edge disjoint paths, if they exist, is shown. This yields an efficient parallel algorithm for testing whether a graph G is k-edge connected. The algorithm runs in $O(k^2 \log n)$ time and uses $nkC(n, kn)$ processors on a CRCW PRAM. Finally, more applications of the disjoint s-t paths algorithm are described.

Key words. graph connectivity, parallel algorithms, disjoint paths

AMS(MOS) subject classifications. 05C38, 05C40, 68Q25, 68R10, 68Q10

1.1. Introduction. Graph connectivity is considered one of the classical subjects in graph theory [Har69], [Ber76], [Eve79], and has many practical applications, e.g., in chip and circuit design, reliability of communication networks, and cluster analysis. Designing efficient parallel algorithms for graph connectivity is clearly a basic problem in parallel computation. Efficient parallel algorithms for testing connectivity [SV82], [CV86], biconnectivity [TV85], triconnectivity [FT88], [FRT89], and four-connectivity [KR87] of graphs have been developed. In this paper we present an efficient parallel algorithm for testing whether a graph is k-vertex connected, for any fixed k. The problem is formally stated as follows. Given an undirected graph $G(V, E)$ and a fixed integer k, test whether G is k-vertex connected. If the graph is not k-vertex connected, find a set of at most $k-1$ vertices whose removal disconnects G.

In order to solve the connectivity problem, we solve the following *disjoint s-t paths* problem. Given an undirected graph G, and two specified vertices s and t, find k-vertex disjoint paths between s and t. If no such paths exist, obtain a set of at most $k-1$ vertices whose removal disconnects s and t. Even [Eve85] proved that in order to test for k-vertex connectivity it is sufficient to check whether there are k-vertex disjoint paths between $n+k^2$ pairs of vertices.

The model of parallel computation used is the Concurrent-Read Concurrent-Write (CRCW) Parallel Random Access Machine (PRAM) [Wyl79]. A PRAM employs p synchronous processors, all having access to a shared memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for both

* Received by the editors August 15, 1989; accepted for publication (in revised form) June 29, 1990. An extended summary of this paper appears in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, October 1989.

† Computer Science Department, Upson Hall, Cornell University, Ithaca, New York 14853. The research of this author was supported by an IBM Graduate Fellowship, National Science Foundation grant DCR 85-52938, and funds from AT&T Bell Laboratories and Sun Microsystems. Part of this research was done while this author was visiting the IBM T. J. Watson Research Center, Yorktown Heights, New York.

‡ IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

read and write purposes. In case several processors attempt to write simultaneously in the same memory location, an arbitrary one succeeds in doing the write. We remark that our algoirthm can also be implemented in weaker PRAM models. The complexity of our algorithm when implemented on such models is the same as the complexity of finding connected components. However, to make the presentation simpler we concentrate on the CRCW PRAM model.

A parallel algorithm is said to have *optimal speedup* if its time-processor product is the same as the time complexity of the best known sequential algorithm for the same problem.

Our parallel algorithm for testing k-vertex connectivity runs in $O(k^2 \log n)$ time using $(n+k^2)kC(n, m)$ processors, where $n=|V|$, $m=|E|$, and $C(n, m)$ is the number of processors required to compute the connected components of a graph with n vertices and m edges in $O(\log n)$ time. For fixed k, our algorithm runs in logarithmic time using $nC(n, m)$ processors. From now on, throughout the paper we assume that k is fixed, and all factors of k will be elided from the complexity bounds.

Using the logarithmic time parallel connectivity algorithm [CV86], the bound for $C(n, m)$ is $(m+n)\alpha(m, n)/\log n$ processors, where $\alpha(m, n)$ is the functional inverse of Ackermann's function, as defined in [Tar75]. Recall that the time complexity of the best known deterministic sequential algorithm for testing k-vertex connectivity is $O(mn)$, for any fixed $k>4$ [Eve75], [Gal80], [BDD⁺82]. Thus, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our k-vertex connectivity algorithm achieve optimal speedup, for any fixed $k>4$. The main component of our parallel k-connectivity algorithm is an efficient parallel algorithm for the disjoint s-t paths problem. This algorithm runs in $O(\log n)$ time using $C(n, m)$ processors. Again, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our parallel disjoint s-t paths algorithm achieve optimal speedup.

The previous best deterministic parallel algorithm for testing k-vertex connectivity, for $k>4$, required $O(\log^2 n)$ time and $nM(n)$ processors (where $M(n)=\Omega(n^2)$ is the number of arithmetic operations required to multiply two $n \times n$ matrices). Similar to our algorithm, this algorithm tests k-vertex connectivity by finding k-vertex disjoint paths between pairs of vertices. The disjoint paths are found using a straightforward parallelization of the sequential algorithm [KMV89]. The parallelization suffers from the problem of being inefficient, requiring $O(\log^2 n)$ time and $M(n)$ processors. A randomized parallel algorithm for testing k-vertex connectivity is given in [LLW86]. It runs in $O(\log^2 n)$ time and uses $n^{2.5}$ processors. Compared to our algorithm, this algorithm uses fewer processors for dense graphs but has the disadvantage of using randomization, and takes $O(\log^2 n)$ time compared to our $O(\log n)$ time bound.

The sequential algorithm for testing connectivity reduces the problem to several flow problems in appropriately defined networks. This reduction yields an efficient sequential algorithm. However, the flow problem does not seem amenable to efficient parallelism since it involves computing reachability in directed graphs. Thus, obtaining an efficient parallel algorithm requires further insights into the problem. In our algorithm we reduce the connectivity problem to a reachability problem in directed graphs, called *bridge graphs*. We are able to exploit the structure of these *bridge graphs* to obtain an efficient parallel reachability algorithm.

This is not the first time that the techniques used for efficient sequential algorithms are not adequate for the respective parallel algorithms. For example, the sequential depth first search (DFS) algorithm is very efficient, and can be used to obtain efficient sequential algorithms for many other problems, such as connectivity, biconnectivity,

and *st*-numbering. However, it seems that DFS is not amenable to efficient parallelism. In order to obtain efficient parallel algorithms for each of these problems, new techniques have to be developed. Examples of these techniques are the Euler tour technique [TV85] and ear decomposition search [MSV86].

Our k-vertex disjoint paths algorithm can be extended to obtain k-edge disjoint paths with the same complexity. The algorithm finds either k-edge disjoint paths, or a set of at most $k-1$ edges whose removal disconnects s and t. This yields a parallel algorithm to test a graph for k-edge connectivity that runs in $O(\log n)$ time using $nC(n, n)$ processors, improving on the previous parallel algorithms in both time and number of processors. The best sequential algorithm for testing k-edge connectivity runs in $O(n^2)$ time, for any fixed $k>2$ [Mat87]. Thus, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our k-edge connectivity algorithm achieve optimal speedup, for any fixed $k>2$.

In addition to its application to connectivity algorithms, the disjoint s-t paths algorithm has several other applications. It can be used in protocols for reliable communication over networks as described in [IR84]. In [KMV89] it is used to obtain efficient parallel algorithms for the two paths problem [Sey80], [Shi80]. We use it to obtain efficient parallel algorithms for the subgraph homeomorphism problem for some fixed pattern graphs.

We remark that for the case $k=2$, the disjoint s-t paths problem can be solved by using an *st*-numbering of the graph. Unfortunately, the technique does not seem to generalize to solving the problem for any $k>2$. For completeness, we describe this solution.

The paper is organized as follows. In § 2 we give some preliminary definitions and background. Before giving a description of the entire algorithm, we describe the simpler case when $k=2$, in § 3. We also describe how to solve the problem when $k=2$ using *st*-numbering. In § 4, we describe the algorithm for finding k vertex disjoint s-t paths. In § 5, we show how to modify the algorithm to get k edge disjoint s-t paths. Finally, § 6 describes some more applications of the disjoint paths algorithm.

Remark. The algorithm for $k=2$, given in § 3, is a special case of the algorithm described in § 4, and is included only to make the presentation clearer. Since § 4 is self-contained, some of the readers might find it useful to skip over § 3.

2. Connectivity and disjoint paths. Let $G(V, E)$ be a simple undirected graph. Without loss of generality we will assume that G is connected. Let s and t be distinct vertices in V that are not connected by an edge. An (s, t) vertex separator is a set of vertices $S_V \subset V$, such that every path from s to t contains at least one vertex from S_V. Define $N(s, t)$ to be the minimum cardinality of an (s, t) vertex separator. By Menger's theorem [Eve79], [BM77] there are exactly $N(s, t)$ vertex disjoint paths between s and t. The vertex connectivity λ_V of G is defined as $\lambda_V = \min\{N(s, t) \mid \{s, t\} \subset V,\ s \neq t,\ (s, t) \notin E\}$. In other words, the vertex connectivity of G is the cardinality of the smallest set of vertices whose removal disconnects G. Note that there are λ_V vertex disjoint paths between any pair of vertices in G.

We define an *articulation vertex* to be a vertex whose removal from G (together with its incident edges) disconnects G. A graph is said to be *biconnected* if its vertex connectivity is at least two. The biconnected components of a graph are its maximally biconnected subgraphs. A graph whose vertex connectivity is at least k, is called *k-vertex connected.*

From the definition of vertex connectivity, it follows that testing whether G is k-vertex connected can be done by checking if $N(s, t) \geqq k$, for all n^2 pairs of vertices.

In [DF56] and [ET75] it was observed that $N(s, t)$ can be computed using max-flow techniques. Given the graph $G(V, E)$, construct a network $N_{s,t}(V, E)$, where each vertex (excluding s and t) and edge has unit capacity. It is not difficult to see that $N(s, t) \geqq k$, if and only if the value of the max-flow from s to t in $N_{s,t}$ is at least k.

In [Eve75] it was proven that to test whether G is k-vertex connected it is sufficient to check whether $N(s, t) \geqq k$, for only $(n + k^2)$ pairs. Here we prove a weaker claim.

Let $V_k = \{v_1, \cdots, v_k\}$ be a set of k vertices of G. (Without loss of generality assume that $|V| > k$.)

CLAIM. *If $N(v_i, u) \geqq k$ for all $i = 1, \cdots, k$ and all $u \in V$, then G is k-vertex connected.*

Proof. Suppose that the graph is not k-vertex connected. This implies that there exists a set of at most $k - 1$ vertices whose removal disconnects G into at least two components C_1 and C_2. Since the size of the separating set is $< k$, at least one vertex v_i is in one of the components. Without loss of generality assume that v_i is in C_1. Let u be a vertex in C_2. Clearly, $N(v_i, u) < k$, yielding a contradiction to the assumption. □

We conclude that we can test whether G is k-vertex connected by running in parallel $(n + k^2)$ copies of an algorithm for obtaining k-vertex disjoint paths. We will describe an algorithm for obtaining k-vertex disjoint paths that takes $O(\log n)$ time and $C(n, m)$ processors. This yields an $O(\log n)$ time algorithm that uses $nC(n, m)$ processors for testing whether a graph is k-vertex connected.

The edge connectivity of G is defined in a similar way. Let s and t be distinct vertices in V. An (s, t) edge separator is a set of edges $S_E \subset E$, such that every path from s to t uses at least one edge from S_E. Define $M(s, t)$ to be the minimum cardinality of an (s, t) edge separator. By Menger's theorem [Eve79], [BM77], there are exactly $M(s, t)$ edge disjoint paths between s and t. The edge connectivity λ_E of G is defined as $\lambda_E = \min \{M(s, t) | \{s, t\} \subset V, s \neq t\}$. In other words, the edge connectivity of G is the cardinality of the smallest set of edges whose removal disconnects G. Note that there are λ_E edge disjoint paths between any pair of vertices in G.

To test whether a graph is k-edge connected we can check whether $M(s, t) \geqq k$, for all n^2 pairs of vertices. As in the vertex case, $M(s, t)$ can also be computed using max-flow techniques [DF56], [ET75]. In the corresponding network $M_{s,t}$ each edge has unit capacity.

It is easy to see that it is sufficient to check whether $M(s, t) \geqq k$, for only n pairs.

CLAIM. *Let v be a vertex in G. If $M(v, u) \geqq k$ for all $u \in V - \{v\}$, then G is k-edge connected.*

Proof. Suppose that the graph is not k-edge connected. This implies that there exists a set of at most $k - 1$ edges whose removal disconnects G into at least two components C_1 and C_2. Without loss of generality assume that v is in C_1. Let u be a vertex in C_2. Clearly, $M(v_i, u) < k$, yielding a contradiction to the assumption. □

Recently, Thurimella [Thu89] observed that to test whether G is k-edge connected it is sufficient to consider a sparse spanning subgraph of G.

Let F_1 be a maximal spanning forest (tree) of G. For $i = 2, \cdots, k$, let F_i be a maximal spanning forest of $G - \bigcup_{j=1}^{i-1} F_j$. Define $H = \bigcup_{i=1}^{k} F_i$. Note that H has $O(kn)$ edges.

THEOREM 2.1 (Thurimella). *The graph G is k-edge connected if and only if its subgraph H is k-edge connected.*

Proof. The *if* direction is trivial. We prove the *only if* direction by contradiction. Assume that G is k-edge connected and H is not. This implies that there exists a set of at most $k - 1$ edges whose removal disconnects H into at least two components (say

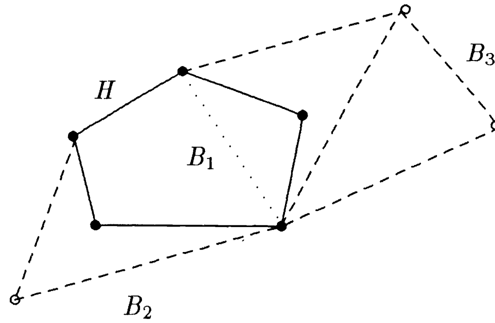


FIG. 1. *An example illustrating bridges.*
——, *edges of H*; - - - - *edges of* B_2 *and* B_3; · · · · *edge of trivial bridge* B_1.

C_1 and C_2). On the other hand, there are at least k edges between C_1 and C_2 in G. Since the edges of the spanning forests are pairwise disjoint, and there are k such forests, at least one such forest, say F_i, does not contain any edge between C_1 and C_2. Thus, in F_i, the components C_1 and C_2 are disconnected. However, in $G-\bigcup_{j=1}^{i-1} F_j$, which is spanned by F_i, there is at least one edge between C_1 and C_2, a contradiction. □

It follows that we can test whether G is k-edge connected by running n copies of an algorithm for obtaining k-edge disjoint paths in parallel on a sparse subgraph H of G, that has $O(kn)$ edges. We construct H by k applications of the parallel connectivity algorithm. This takes $O(\log n)$ time and $C(n, m)$ processors. We will describe a parallel algorithm for obtaining k-edge disjoint paths in a graph with n vertices and m edges, that runs in $O(\log n)$ time and $C(n, m)$ processors. This yields an $O(\log n)$ time algorithm that uses $nC(n, n)$ processors for testing whether a graph is k-edge connected.

We conclude the preliminaries with the following definition of *bridges*. Given a graph $G(V, E)$, let H be a subgraph of G, and let e and f be edges of G not in H. Define the equivalence relation $=_H$ by $e =_H f$, if and only if there is a path in G that includes e and f and has no internal vertices in common with $V(H)$. The subgraphs induced by the edges of the equivalence classes of $E(G)-E(H)$ under $=_H$ are called the *bridges* of G relative to H. The *attachment vertices* of a bridge B relative to H are the vertices in $V(B)\cap V(H)$.

In Fig. 1, H is a simple cycle in G. The bridge B_1 is a trivial bridge consisting of only one edge, the two other bridges B_2 and B_3 are nontrivial.

3. Finding two vertex disjoint paths. Let $G(V, E)$ be an undirected graph, with s and t vertices that are not connected by an edge. In this section, we develop a parallel algorithm for finding either two vertex disjoint paths between s and t, or an articulation vertex separating s and t. We will subsequently show how this algorithm can be generalized to finding k-vertex disjoint s-t paths, with the same complexity bounds.

High level description. The algorithm consists of five steps.

Step 1. Find a path P_1 from s to t.

Step 2. Decompose G into its bridges relative to the path P_1.

If there is a single bridge that has both s and t as its attachment vertices, then a path P_2 between s and t in this bridge is vertex disjoint from P_1, yielding the two disjoint paths. Suppose that there is no such bridge.

Define a linear order on the attachment vertices according to their position on P_1. An attachment vertex a is said to be less than an attachment vertex b, if a is to

the left of b on P_1. (We assume that s is the leftmost vertex on P_1 and t is the rightmost.) For each bridge B_i, define l_i to be leftmost attachment vertex, i.e., the attachment vertex that is closest to s on P_1. Similarly, define r_i to be its rightmost attachment vertex; i.e., the attachment vertex that is furthest from s on P_1.

Step 3. Construct a new directed graph $G_B(V_B, E_B)$, called the *bridge graph*, as follows:

$$V_B = \{\beta_i \mid B_i \text{ is a bridge}\} \cup \{s, t\}.$$

The set of edges E_B is defined as follows. The vertex s has an outgoing edge to a vertex β_j, if $l_j = s$, and r_j is furthest from s, among all bridges B_z, with $l_z = s$. A vertex β_i has an outgoing edge to t if $r_i = t$. If $r_i \neq t$, then we define an outgoing edge from β_i as follows: each vertex β_i has an outgoing edge to a vertex β_j if r_j is the rightmost vertex on P_1 among the attachment vertices of bridges whose leftmost attachment vertex is to the left of r_i, and if $r_j > r_i$.

Remark. When the maximum is achieved for more than one bridge, any one is chosen arbitrarily. This implies that the outdegree of every vertex of G_B is at most one.

Step 4. Find a (directed) path from s to t in G_B, if one exists.

We prove below that such a path in G_B exists if and only if there are two vertex disjoint paths between s and t. We also show how to construct these two paths, given the path in G_B.

Step 5. If a path from s to t in G_B was found, construct two vertex disjoint paths between s and t. (The construction is given below.) Otherwise (there is no path from s to t in G_B), if there is no bridge with s as an attachment vertex, then the neighbor of s on P_1 separates s and t, else, consider all the vertices reachable from s in G_B. Recall that each such vertex corresponds to a bridge of G relative to P_1. Let w be the attachment vertex of one of these bridges which is furthest from s. The vertex w separates s from t.

Correctness. To prove the correctness of the algorithm we prove the following theorem.

THEOREM 3.1. *There are two vertex disjoint paths between s and t in G if and only if there is a directed path from s to t in G_B.*

Proof. The *if* direction. Suppose that there is a directed path $P_B = s, \beta_{x_1}, \cdots, \beta_{x_a}, t$ from s to t in G_B. To prove that there are two vertex disjoint paths from s to t in G, we show that the value of the max-flow in the network $N_{s,t}$ is at least two.

Observe that the path P_1 corresponds to a path in $N_{s,t}$ from s to t. Using P_1 we can push one unit of flow from s to t. Our goal is to show that P_B defines an augmenting path in $N_{s,t}$ given the flow corresponding to P_1.

For each vertex β_{x_i} on P_B, let R_{x_i} be the path in B_{x_i} from l_{x_i} to r_{x_i}. (Note that $l_{x_1} = s$ and $r_{x_a} = t$.)

Given a path P, let $P[l; r]$ denote its segment from l to r. We define P_2 to be a path from s to t in G as follows:

$$P_2 = R_{x_1}; P_1[r_{x_1}; l_{x_2}]; R_{x_2}; \cdots; R_{x_{a-1}}; P_1[r_{x_{a-1}}; l_{x_a}]; R_{x_a}.$$

Path P_2 consists of two kinds of segments: segments belonging to bridges and segments belonging to P_1 that "reverse" on P_1 (see Fig. 2). (We view P_1 as directed from s to t.)

We claim that P_2 corresponds to an augmenting path in $N_{s,t}$, yielding a flow of two units (see Fig. 3). This follows from the following lemma.

LEMMA 3.2. *After pushing one unit of flow along P_2, the outgoing flow from each vertex (excluding s) and the incoming flow to each vertex (excluding t) is at most one.*

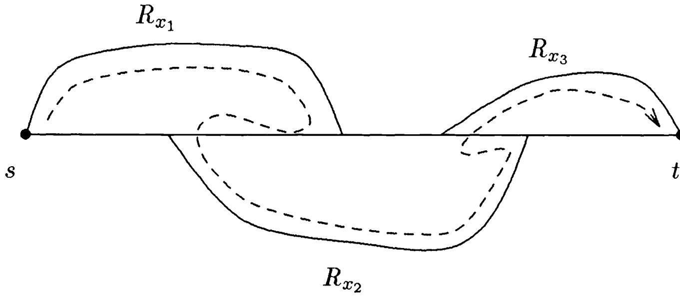


FIG. 2. *Path P_2.*

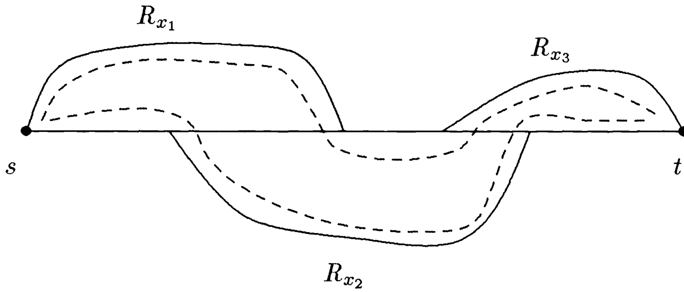


FIG. 3. *Obtaining two vertex disjoint s-t paths.*

Proof. Before pushing the flow along P_2 we had a legal flow of value one. Clearly, by pushing flow along P_2 we change the flow only for the vertices of P_2; thus we may consider only these vertices. Consider an internal vertex v on P_2. If v is not on P_1 (that is, it is an internal vertex of some R_{x_i}), then its incoming and outgoing flow is one.

Suppose that v is a vertex on $P_1[r_{x_i}; l_{x_{i+1}}]$. From the definition of G_B it follows that $l_{x_{i+1}} < r_{x_i}$, for $1 \leqq i < a$. Clearly, $l_{x_{i+2}} \geqq r_{x_i}$, for $1 \leqq i \leqq a-2$, since otherwise β_{x_i} would have an outgoing edge to $\beta_{x_{i+2}}$. We observe that the path P_2 has the following *monotonicity property*: $l_{x_2} < r_{x_1} \leqq l_{x_3} < r_{x_2} \leqq \cdots < r_{x_{a-2}} \leqq l_{x_a} < r_{x_{a-1}}$. This implies that if v appears more that once on P_2, then it appears twice, and $v = r_{x_i} = l_{x_{i+2}}$, for some $1 \leqq i \leqq a-2$ (see Fig. 4). Also note that P_2 is (edge) simple (i.e., it does not repeat any edges).

We distinguish between three cases: (1) The vertex v is not an endpoint of any path R_{x_i}. (2) The vertex v is a right endpoint of R_{x_i} and a left endpoint of $R_{x_{i+2}}$ (i.e., $v = r_{x_i} = l_{x_{i+2}}$). (3) The vertex v is an endpoint of one path R_{x_i} (i.e., either $v = l_{x_i}$ or $v = r_{x_i}$). (The figures describing all the cases are given in Fig. 9.) The proofs for each

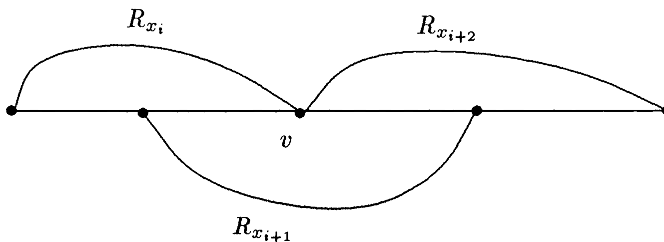


FIG. 4. *Vertex v occurs on P_2 twice.*

of these cases are given in the proof of Lemma 4.2 in the next section. (Lemma 4.2 is the version of this lemma for an arbitrary k.) □

The *only if* direction (of Theorem 3.1). Assume that there is no path from s to t in G_B. If there is no bridge with s as an attachment vertex, then the neighbor of s on P_1 separates s and t. Consider all the vertices reachable from s in G_B. Recall that each such vertex corresponds to a bridge of G relative to P_1. Let w be the attachment vertex of one of these bridges, say B_i, which is furthest from s. We claim that w separates s and t. Remove w and all its incident edges from G, and assume that there exists a path Q from s to t in the remaining graph.

Consider the vertices of Q that are also on P_1 in the order of their appearance on Q. Observe that there must be two such successive vertices w_1 and w_2 such that $w_1 < w$ and $w_2 > w$. Clearly, the subpath of Q from w_1 to w_2 does not contain any vertex of P_1. Hence, there exists a bridge B_j relative to P_1 such that $l_j < w$ and $r_j > w$, contradicting the definition of w. □

Implementation and complexity. We show how to implement the algorithm in $O(\log n)$ time and $C(n, m)$ processors on a CRCW PRAM. In the following discussion we will assume that G is represented by its adjacency list.

Step 1. We find the path P_1 from s to t, by computing a spanning tree of G, and following the unique path from s to t in this tree. This is done in $O(\log n)$ time and $C(n, m)$ processors using a logarithmic time parallel graph connectivity algorithm (e.g., [CV86]).

Steps 2, 3. The implementation of these steps is analogous to the implementation of Steps 2 and 3 of the k disjoint paths algorithm given in the next section. The detailed description of these steps is postponed to that section.

Step 4. Observe that the outdegree of each vertex in G_B is at most one. Observe also that G_B is acyclic, since $(\beta_x \to \beta_y)$ in G_B implies that $r_x < r_y$. We conclude that G_B is a rooted forest (where the edges are directed towards the root). Since the outdegree of t is zero, t is the root of some tree in the forest. Thus, there exists a path from s to t if and only if s is the tree rooted at t.

We can check if s is in the tree rooted at t by applying the Euler tour technique of [TV85]. However, to apply this technique we need the full adjacency list of the forest, i.e., we need to compute the list of incoming edges to each vertex β_i. To do this we use the following observation. Consider two edges $(\beta_x \to \beta_i)$ and $(\beta_y \to \beta_i)$ incoming to β_i. Suppose that $r_x < r_y$. Then, all the bridges with rightmost attachment vertex between r_x and r_y have an outgoing edge to β_i.

For a bridge B_i, define the *rightmost edge* to be the edge of B_i whose endpoint is the rightmost attachment of B_i. Consider the concatenation of the adjacency lists (in G) of all the vertices on P_1. Compact this list to include only the rightmost edges of bridges. It follows from the above observation that the rightmost edges of all the bridges whose corresponding outgoing edges in G_B point to the same vertex in G_B are consecutive in this concatenated list. Thus, the list of incoming edges of all the vertices in G_B can be computed using the algorithms for list ranking and prefix sums in $O(\log n)$ time and $(m+n)/\log n$ processors [CV86], [AM88], [LF80].

Step 5. If there is no path from s to t in G_B, the separating vertex w is the rightmost attachment vertex of the bridge corresponding to the root of the tree containing s. This rightmost attachment can be found in $O(\log n)$ time with $n/\log n$ processors using the same technique used in Step. 4.

If there is a path from s to t in G_B, the two vertex disjoint paths Q_1 and Q_2 can be constructed by following the two flow paths from s to t. We define Q_1 and Q_2

recursively. First, we define the prefix of each of these paths and then for each edge on these paths we define its successive edge. The prefix of Q_1 is $P_1[s; l_{x_2}]$. The prefix of Q_2 is R_{x_1}. The successor edge for an edge $(u \to v)$, $v \neq t$, is the edge along which the unit flow leaves v. Updating the incoming and outgoing flow edges through each vertex is easily accomplished after computing P_2. Each vertex of P_2 that is also on P_1 independently adjusts its incoming and outgoing flow edges. For some vertices the flow through them is completely canceled, and they are neither on Q_1 nor Q_2. In this way, we obtain the two paths as linked lists that can be ranked in $O(\log n)$ time using $n/\log n$ processors [CV86], [AM88], [CV88].

To conclude this section, we remark that two disjoint s-t paths can be obtained from an st-numbering of the graph. This alternative algorithm has the advantage that after computing this numbering once, we can find two vertex disjoint paths between *any* pair of vertices. However, it seems that this algorithm cannot be generalized to obtain k-vertex disjoint paths, for $k > 2$.

For completeness we describe this algorithm briefly. Recall the definition of an st-numbering [LEC67], [ET76]. Let $G(V, E)$ be an undirected graph and let (s^*, t^*) be an edge in G. An st-numbering of G is a 1-1 function $f: V \to \{1, \cdots, n\}$ with the following properties: (1) $f(s^*) = 1$; (2) $f(t^*) = n$; (3) any vertex $v \in V - \{s^*, t^*\}$ has at least one adjacent vertex u with $f(u) < f(v)$ and at least one adjacent vertex w with $f(w) > f(v)$. It is not difficult to see that a graph is biconnected if and only if it has an st-numbering starting from any edge (s^*, t^*) [LEC67].

The algorithm consists of two stages: a preprocessing stage, in which an st-numbering is computed, and a processing stage which uses this numbering to obtain two vertex disjoint paths.

In the preprocessing path we decompose G into its biconnected components, and compute an st-numbering for each biconnected component.

We now show how to find two vertex disjoint paths between s and t. First, we check to ensure that s and t are in the same biconnected component.

We construct a simple cycle C_s that contains both s and s^* by concatenating two vertex disjoint paths from s to s^*. One path is computed by starting at s and following a sequence of vertices with decreasing st numbers, until we reach s^*. We are guaranteed to reach s^* since each vertex (excluding s^*) has an adjacent vertex with a smaller number. Similarly, we construct a second path by starting at s and following a sequence of vertices with increasing st numbers until we reach t^*, and we then use the edge (s^*, t^*) to reach s^*. If t is on C_s then the two vertex disjoint paths from s to t are obtained from C_s. Otherwise, we compute a simple cycle C_t that contains both t and s^* in a similar way. Note that both C_t and C_s contain the edge (s^*, t^*).

Let w_1 be the first vertex on C_s to the left of s that is also on C_t, and let w_2 be the first vertex on C_s to the right of s that is also on C_t. It is easy to see that the parts of C_s and C_t from w_1 to w_2 define a simple cycle that contains both s and t, yielding two vertex disjoint paths between s and t. (See Fig. 5.)

The preprocessing stage of the algorithm can be done in $O(\log n)$ time and $C(n, m)$ processors on a CRCW PRAM, using the parallel algorithms for testing biconnectivity [TV85] and computing an st-numbering [MSV86]. Given this preprocessing the rest of the computation can be done in $O(\log n)$ time with $n/\log n$ processors, using the optimal logarithmic time parallel algorithm for list ranking [CV86], [AM88].

4. Finding k vertex disjoint paths. Let $G(V, E)$ be an undirected graph, with s and t two specified vertices not connected by an edge. In this section we describe a parallel algorithm for finding either k-vertex disjoint paths between s and t, or a set

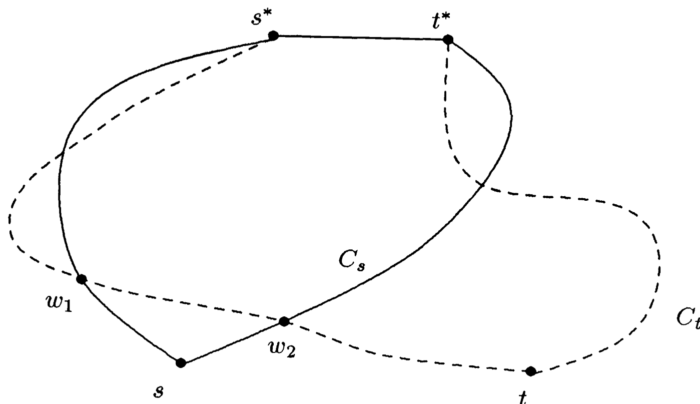


FIG. 5. *Obtaining two vertex disjoint s-t paths.*

of at most $k-1$ vertices whose removal separates s from t. This algorithm is a generalization of the algorithm for obtaining two disjoint s-t paths given in the preceding section. It runs in $O(\log n)$ time and uses $C(n, m)$ processors.

High level description. The algorithm consists of five steps.

Step 1. If $k=1$, find a path from s to t using a spanning tree of the graph. Otherwise, recursively find $k-1$ vertex disjoint paths from s to t, if they exist. If not, output a separating set of size $\leqq k-2$.

Suppose that $k-1$ paths, $P_1, \cdots, P_{k-1}$ were found.

Step 2. Decompose the graph into its bridges relative to the subgraph $P_1 \cup P_2 \cup \cdots \cup P_{k-1}$.

If there is a bridge that has both s and t as its attachment vertices, then a path P_k between s and t in this bridge is vertex disjoint from $P_1, \cdots, P_{k-1}$, yielding the k disjoint paths. Suppose that there is no such bridge.

For Step 3 we need the following definitions.

DEFINITION 1. Suppose that both attachment vertices a and b are on a path P_j. The vertex a is said to be less than b, if a is to the left of b on P_j. (We assume that s is the leftmost vertex on P_j, and t the rightmost.)

This defines a linear order on the attachment vertices lying on a path P_j.

DEFINITION 2. For a bridge B_i and a path P_j, let l_i^j be the leftmost attachment vertex of B_i on P_j, and r_i^j be the rightmost attachment vertex of B_i on P_j. If B_i has no attachment vertices on P_j, l_i^j and r_i^j are undefined.

Step 3. Construct a new directed graph $G_B(V_B, E_B)$, called the *bridge graph*, defined as follows:

$$V_B = \{\beta_i \mid B_i \text{ is a bridge}\} \cup \{s, t\}.$$

The edges in E_B are the union of $k+1$ sets.

1. The set $S=\{e_1, \cdots, e_{k-1}\}$ of edges outgoing from s:

$$e_j = \{s \to \beta_x \mid r_x^j = \max_z r_z^j \text{ s.t. } (l_z^j = s) \wedge r_x^j > s\}.$$

   In words, the edge e_j is $(s \to \beta_x)$ if r_x^j is the rightmost vertex on P_j, to the right of s, among the attachment vertices of bridges whose leftmost attachment is s. Note that if $(s \to \beta_x)$ is an edge then $l_x^j = s$, for *all* $1 \leqq j \leqq k-1$.

2. The set T of edges incoming to t:

$$T = \{\beta_x \to t \mid r_x^j = t, \text{for some } 1 \leqq j \leqq k-1\}.$$

In words, the set T consists of edges $(\beta_j \to t)$, for all bridges B_j whose rightmost attachment is t. Note that if $(\beta_x \to t) \in T$ then $r_x^j = t$, for *all* $1 \leq j \leq k-1$.

3. The edges between vertices corresponding to bridges. These edges are partitioned into sets $D_1, \cdots, D_{k-1}$. We add an edge $(\beta_i \to \beta_x)$ to D_j, if it is possible to "move" from bridge B_i to B_x by "reversing" along some path P_y, and r_x^j is the furthest we can move (from B_i) along path P_j (see Fig. 6). More formally, the edge $(\beta_i \to \beta_x) \in D_j$ if
   (a) For some path P_y $(1 \leq y \leq k-1)$, we have $(s < l_x^y < r_i^y)$.
   (b) We have $(r_x^j > r_i^j)$. (This condition is considered satisfied in case B_i has no attachment vertex on P_j.)
   (c) There is no β_z, such that β_z satisfies the above two conditions, and $r_z^j > r_x^j$.
   In other words, to determine the outgoing edge from β_i in the set D_j, consider all bridges whose leftmost attachment vertex on some P_y is to the left of r_i^y (and is not s). The edge $(\beta_i \to \beta_x)$ is added to D_j if B_x has the rightmost attachment on P_j, among all bridges in the set, and this attachment vertex r_x^j is to the right of r_i^j. Ties are broken lexicographically, i.e., if β_x and β_y are both candidates for the outgoing edge from β_i, we choose β_x if $x < y$. (Each bridge can be uniquely labeled by the index of the lowest numbered edge it contains.)

The graph G_B is illustrated by an example in Fig. 7.

Note that each vertex, except s and t, has at most one outgoing edge belonging to each set D_j. Since there are $k-1$ such sets, the outdegree of each such vertex is at most $k-1$. It is easy to see also that the outdegree of s is at most $k-1$ and the outdegree of t is zero.

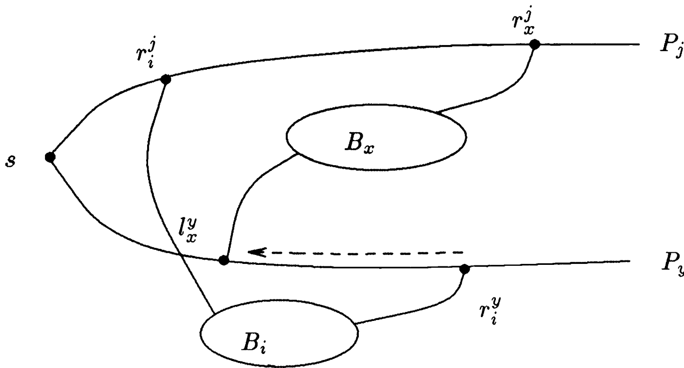


FIG. 6. *The edge* $(\beta_i \to \beta_x)$ *is added to* D_j.

Our algorithm is based on two ideas. First, we show that there exists a directed path from s to t in G_B if and only if there are k vertex disjoint paths between s and t in G. Moreover, given the path in G_B, the k disjoint s-t paths in G can be constructed. Second, by exploiting the structure of G_B, we show how to find a path from s to t (if one exists) efficiently.

Step 4. Find a (directed) path from s to t in G_B, if one exists.

Step 5. If a path from s to t in G_B was found, construct the k vertex disjoint paths between s and t. Suppose that there is no path from s to t in G_B. Consider all the vertices β_i reachable from s in G_B. Recall that each such vertex corresponds to a

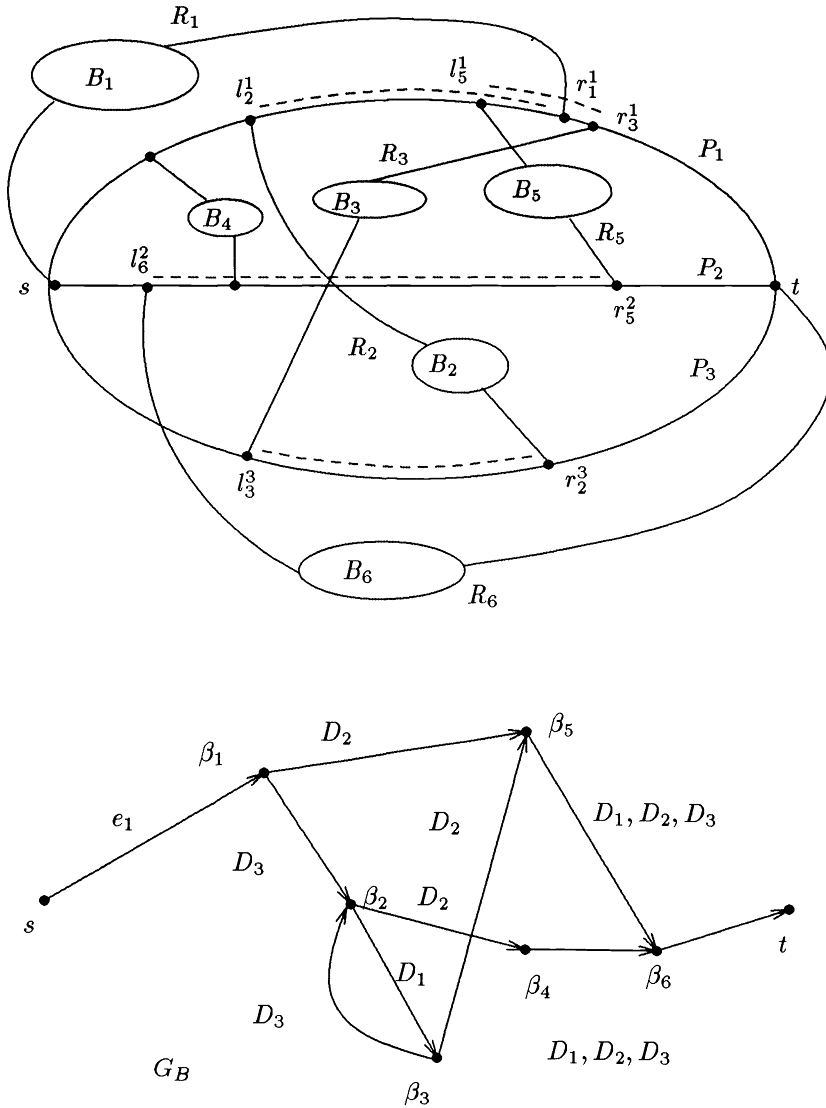


FIG. 7. *The path P_k corresponding to $P_B = s, \beta_1, \beta_2, \beta_3, \beta_5, \beta_6, t$ (in G_B).*

bridge B_i of G. Let w_j be the attachment vertex of one of these bridges which is furthest from s on P_j. In case there is no attachment vertex of any of these bridges on P_j, then w_j is chosen to be the neighbor of s on P_j. The set $\{w_1, \cdots, w_{k-1}\}$ separates s from t.

Correctness. Suppose that there are $k-1$ vertex disjoint paths between s and t. By our induction hypothesis these paths are found in Step 1. The base case is when $k=1$. In this case the path P_1 from s to t is found by computing a spanning tree of G, and following the unique path from s to t in this tree. To prove the inductive step we prove the following theorem.

THEOREM 4.1. *There are k vertex disjoint paths between s and t in G if and only if there is a directed path from s to t in G_B.*

Proof. The *if* direction: Suppose that there is a directed path $P_B = s, \beta_{x_1}, \cdots, \beta_{x_a}, t$ from s to t in G_B. Assume that P_B is (vertex) simple (i.e., no vertex appears twice in P_B).

To prove that there are k vertex disjoint paths from s to t in G, we show that the value of the max-flow in the network $N_{s,t}$ is at least k. Observe that the paths $P_1, \cdots, P_{k-1}$ correspond to $k-1$ vertex disjoint paths in $N_{s,t}$ from s to t. Using these paths we can push $k-1$ flow units from s to t. Our goal is to show that P_B defines an augmenting path in $N_{s,t}$ given this flow.

For an edge $(\beta_i \rightarrow \beta_x)$ in G_B, define the *type* of $(\beta_i \rightarrow \beta_x)$ to be y, if $r^y_i > l^y_x$. For example, in Fig. 6, the type of the outgoing edge from β_i is y (since the "reversal" is done on path P_y). Note that by the definition of G_B, the type is always defined. In case there are several such y's any one may be chosen to be the type of the edge. Given the path $P_B = s, \beta_{x_1}, \cdots, \beta_{x_a}, t$, let i_b be the type of the outgoing edge from β_{x_b}, for $1 \leqq b \leqq a-1$.

For each vertex β_{x_b} on P_B, define a path R_{x_b} in B_{x_b} as follows. For B_{x_1}, R_{x_1} is from s to $r^{i_1}_{x_1}$, where i_1 is the type of the outgoing edge from β_{x_1}. For B_{x_a}, R_{x_a} is from $l^{i_{a-1}}_{x_a}$ to t, where i_{a-1} is the type of the outgoing edge from $\beta_{x_{a-1}}$. For $B_{x_b}, 1 < b < a$, R_{x_b} is from $l^{i_{b-1}}_{x_b}$ to $r^{i_b}_{x_b}$, where i_{b-1} is the type of the outgoing edge from $\beta_{x_{b-1}}$ and i_b is the type of the outgoing edge from β_{x_b} in P_B. See Fig. 7 for an example. In this example $a = 5$, and the paths $P_{i_1}, P_{i_2}, P_{i_3}, P_{i_4}$ are P_1, P_3, P_1, P_2, respectively. The values of x_1, x_2, x_3, x_4, x_5 are 1, 2, 3, 5, 6, respectively.

Define a path P_k from s to t in G

$$P_k = R_{x_1}; P_{i_1}[r^{i_1}_{x_1}; l^{i_1}_{x_2}]; R_{x_2}; \cdots; R_{x_{a-1}}; P_{i_{a-1}}[r^{i_{a-1}}_{x_{a-1}}; l^{i_{a-1}}_{x_a}]; R_{x_a}.$$

Path P_k consists of the following two kinds of segments: segments belonging to bridges, and segments belonging to paths P_i that "reverse" on P_i. (We view each P_i as directed from s to t.)

The path P_k is not necessarily (edge) simple as shown in Fig. 7. (The figure illustrates only the leftmost and rightmost attachment vertices of each bridge.) This happens when some segments of P_k that "reverse" on some P_i overlap. We claim that, given P_k, we can construct an (edge) simple path from s to t that consists of two kinds of segments: segments belonging to bridges, and segments that "reverse" on some P_i. This is achieved by a "pruning" step on the path P_k. Moreover, we can obtain a "pruned" path P_k with the following *monotonicity property.* If $b < c$, and both $P_i[r^i_{x_b}; l^i_{x_{b+1}}] P_i[r^i_{x_c}; l^i_{x_{c+1}}]$ are segments of the edge simple path P_k, then $l^i_{x_{b+1}} < r^i_{x_b} \leqq l^i_{x_{c+1}} < r^i_{x_c}$. (In other words, as we move on the "pruned" P_k from s to t the segments of P_k that are "reverse" segments are ordered on P_i.)

The "pruning" step on P_k is done as follows. Consider the segment $P_i[r^i_{x_b}; l^i_{x_{b+1}}]$. Let d be the maximum index such that $d > b$, and $l^i_{x_d} < r^i_{x_b}$ on P_i. (Note that d is always defined, since $d = b+1$ is a possible candidate.) In this case we "prune" the path P_k, and replace the subpath of P_k from $r^i_{x_b}$ to $l^i_{x_d}$ by the segment $P_i[r^i_{x_b}; l^i_{x_d}]$ (see Fig. 8). The simple path is constructed by following the chain from s to t in the resulting graph. Let

$$P_k = R_{x_1}; P_{i_1}[r^{i_1}_{x_1}; l^{i_1}_{x_2}]; R_{x_2}; \cdots; R_{x_{a-1}}; P_{i_{a-1}}[r^{i_{a-1}}_{x_{a-1}}; l^{i_{a-1}}_{x_a}]; R_{x_a}$$

denote the resulting (edge) simple path.

We claim that P_k corresponds to an augmenting path in $N_{s,t}$. This follows from the following lemma.

LEMMA 4.2. *After pushing one unit of flow along P_k, the outgoing flow from each vertex (excluding s) and the incoming flow to each vertex (excluding t) is at most one.*

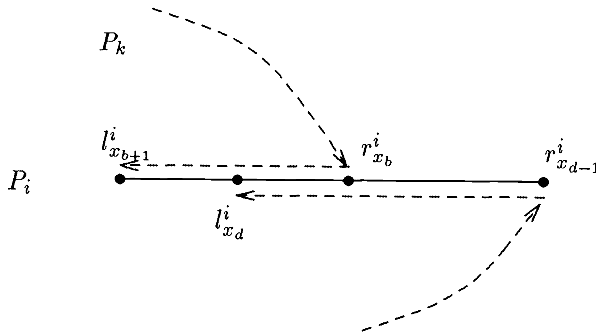


FIG. 8. *We prune the path so that P_k goes from B_{x_b} to B_{x_d}.*

Proof. Before pushing the flow along P_k we had a legal flow of value $k-1$. Clearly, by pushing flow along P_k we change the flow only for the vertices on P_k, thus we may consider only these vertices. Consider an internal vertex v on P_k. If v is not on any P_i, $i<k$ (that is, it is an internal vertex on some R_{x_i}), then its incoming and outgoing flow is one.

Suppose that v is an internal vertex on some P_i. Let $(u \to v)$ and $(v \to w)$ be the incoming and outgoing edges of v on P_i, respectively. We have three cases. (The figures describing these cases are given in Fig. 9.)

Case 1. The vertex v is not an endpoint of any path R_{x_b}. In this case v appears only once on P_k. Observe that on P_k we have the edges $(w \to v)$ and $(v \to u)$ (that is, the reverse of the corresponding edges on P_i). After pushing one unit of flow along P_k, the incoming and outgoing flow of v is zero.

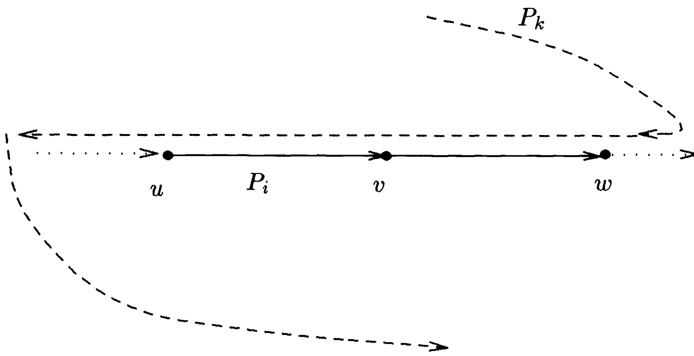
Case 2. The vertex v is a right endpoint of R_{x_b} and a left endpoint of $R_{x_{c+1}}$ (i.e., $v = r^i_{x_b} = l^i_{x_{c+1}}$), for some $b<c$. Both $P_i[r^i_{x_b}; l^i_{x_{b+1}}]$ and $P_i[r^i_{x_c}; l^i_{x_{c+1}}]$ are segments of the edge simple path P_k. Since $r^i_{x_b} = l^i_{x_{c+1}}$, by the monotonicity property there is no d, $(b<d<c)$ such that R_{x_d} has an endpoint on P_i. It follows that the vertex v appears exactly twice on P_k. In the first appearance, the incoming edge of P_k is the last edge of R_{x_b}, and the outgoing edge is $(v \to u)$ (reverse of the edge on P_i). In the second appearance on P_k, the incoming edge is $(w \to v)$ and the outgoing edge is the first edge on $R_{x_{c+1}}$. After pushing one unit of flow along P_k, the incoming flow to v is one (from R_{x_b}), and the outgoing flow from v is one (towards $R_{x_{c+1}}$).

Case 3. The vertex v is an endpoint of one path R_{x_b} (i.e., either $v = l^i_{x_b}$ or $v = r^i_{x_b}$). We prove it only for the case $v = l^i_{x_b}$; the proof for the case $v = r^i_{x_b}$ is analogous. The vertex v appears only once on P_k. Observe that on P_k we have the edges $(w \to v)$ and the outgoing edge from v on R_{x_b}. After pushing a unit flow along P_k the incoming flow to v is one (from u) and the outgoing flow from v is one (towards R_{x_b}).

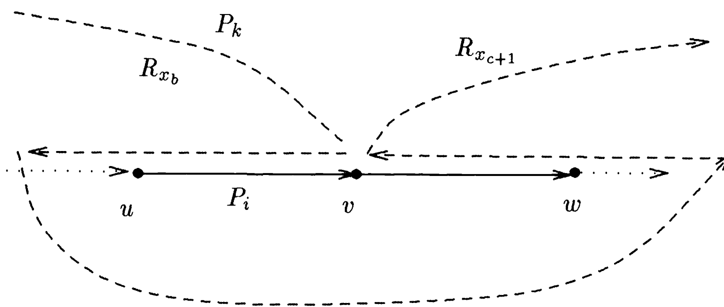
This concludes the proof of the *if* direction of Theorem 4.1. □

The *only if* direction: Assume that there is no path from s to t in G_B. Consider all the vertices β_i reachable from s in G_B. Recall that each such vertex corresponds to a bridge B_i of G. Let w_j be the attachment vertex of one of these bridges which is furthest from s on P_j. In case there is no attachment vertex of any of these bridges on P_j, then w_j is chosen to be the neighbor of s on P_j. We claim that $\{w_1, \cdots, w_{k-1}\}$ separate s from t. To see that, remove the vertices $\{w_1, \cdots, w_{k-1}\}$ and all their incident edges from G, and assume that there exists a path Q from s to t in the resulting subgraph.

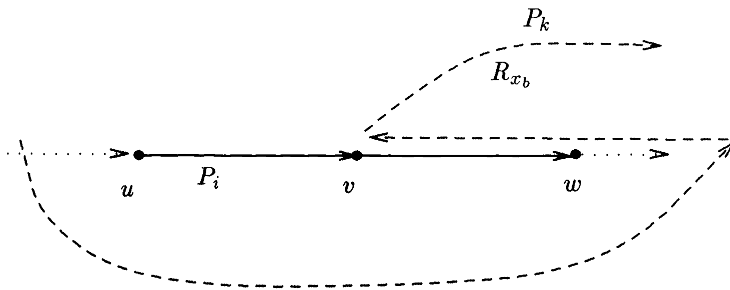
Consider the vertices of Q that are also on one of the paths P_j, $1 \leqq j \leqq k-1$, in the order of their appearance on Q. Observe that there must be two successive vertices x_1 and x_2 such that: (i) x_1 is on some P_j and $x_1 < w_j$, (ii) x_2 is on some $P_{j'}$ and $x_2 > w_{j'}$.



(a) Case 1.



(b) Case 2.



(c) Case 3.

FIG 9. *Figures to illustrate all three cases.*

Clearly, the subpath of Q from x_1 to x_2 does not contain any vertex of $P_1, \cdots, P_{k-1}$. Hence, there exists a bridge B_i such that $l_i^j < w_j$ and $r_i^{j'} > w_{j'}$. This implies that β_i is reachable from s in G_B, contradicting the definition of $w_{j'}$. □

Implementation and complexity. We show how to implement the algorithm in $O(\log n)$ time using $C(n, m)$ processors on a CRCW PRAM, for any fixed k. In the following discussion we will assume that G is represented by its adjacency list.

Step 1. This is the recursive step.

Step 2. The bridges of G relative to the subgraph $P_1 \cup P_2 \cup \cdots \cup P_{k-1}$, and the attachment vertices of each bridge are obtained by computing the spanning forest of the subgraph of G induced by the vertices in $V(G) - V(P_1 \cup P_2 \cup \cdots \cup P_{k-1})$. By adding the edges incident to the vertices in $V(P_1 \cup P_2 \cup \cdots \cup P_{k-1})$ (that are not in $P_1 \cup P_2 \cup \cdots \cup P_{k-1}$) we can obtain all the trivial bridges, as well as the edges that go from vertices on the P_i paths to internal vertices of bridges. This is done in $O(\log n)$ time and $C(n, m)$ processors using a logarithmic time parallel graph connectivity algorithm (e.g., [CV86]).

Step 3. We show how to compute the edges of G_B. Recall that $(\beta_x \to \beta_y) \in D_i$, if r_y^i is the rightmost vertex on P_i, to the right of r_x^i, among the rightmost attachments of bridges whose leftmost attachment vertex on some P_z is to the left of r_x^z (and is not s). We do the computation in $k-1$ phases. Phase i consists of three substeps, as follows:

1. Each vertex w on $P_1 \cup P_2 \cup \cdots \cup P_{k-1}$, considers all the bridges that are attached to it. Among them it selects the bridge whose rightmost attachment vertex on P_i is furthest from s. Denote this bridge by $M_i(w)$. This can be done, for all the vertices on $P_1 \cup P_2 \cup \cdots \cup P_{k-1}$, in $O(\log n)$ time and $(m+n)/\log n$ processors using the optimal logarithmic time algorithm for finding the maximum [SV81].
2. Associate with each vertex w the rightmost attachment vertex of $M_i(w)$ on P_i. (For convenience we refer to this attachment vertex as $M_i(w)$ as well.) For each vertex v on a path P_j, compute the prefix maximum $M_i^*(v) = \max_{w<v} M_i(w)$ (i.e., the maximum of $M_i(w)$ over all vertices w to the left of v on P_j). This can be done, for all the vertices in $O(\log n)$ time with $m/\log n$ processors, using the optimal logarithmic time algorithm for computing prefix maxima. This algorithm can be deduced from the general parallel algorithm for prefix computations [LF80].
3. Given the prefix maxima, the outgoing edge from the set D_i of a bridge B_x can be computed by taking the maximum over the set $\{M_i^*(r_x^1), \cdots, M_i^*(r_x^{k-1})\}$. This can be done, for all the bridges in $O(\log n)$ time with $mk/\log n$ processors, using the optimal logarithmic time algorithm for finding the maximum [SV81].

Step 4. We have to find a path from s to t in the directed graph G_B. In general, it is not known if s-t paths in directed graphs can be obtained in $O(\log n)$ time and $C(n, m)$ processors. We show how to find such a path in $O(k \log n)$ time using $k(m+n)/\log n$ processors, by exploiting the structure of G_B.

For $1 \leqq j \leqq k-1$, let F_j be the subgraph of G_B induced by the edges from the set $D_j \cup \{e_j\}$. (Recall that e_j is an outgoing edge from s.) Observe that the outdegree of each vertex in F_j is at most one. Observe also that F_j is acyclic, since $(\beta_x \to \beta_y)$ in F_j implies that $r_x^j < r_y^j$ (and also, there is no edge incoming to s). We conclude that F_j is a rooted forest (where the edges are directed towards the root).

Define a "shortcutting" operation *over* the edges from $D_j \cup \{e_j\}$. In the "shortcutting" the outgoing edges of all vertices with outgoing edges in $D_j \cup \{e_j\}$ are updated. Consider such a vertex β_y. (We assume that this vertex is not s. Later we describe how the updating is done for s.)

First, the outgoing edge from β_y from the set D_j is deleted. Let β_z be the root of the tree in F_j containing β_y. Note that β_z has no outgoing edge from D_j. If β_z has no outgoing edges in G_B, then this completes the "shortcutting." Suppose that β_z has outgoing edges. If β_z has an outgoing edge to t, then we add an edge from β_y to t (in case such an edge does not exist). Suppose that β_z has no outgoing edge to t. For $a = 1, \cdots, k-1$, $a \neq j$, define β_{z_a} to be the vertex whose attachment point on P_a is the

rightmost among the attachment points of all vertices with incoming edges from β_z. That is, $r^a_{z_a} \geqq r^a_{z'}$, for all $\beta_{z'}$ such that $(\beta_z \to \beta_{z'}) \in E_B$. Observe that β_{z_a} is defined if β_z has no outgoing edges, and that usually $(\beta_z \to \beta_{z_a})$ is the outgoing edge from the set D_a. This is not the case only when $r^a_{z_a} \leqq r^a_z$. If β_y has an outgoing edge to t, then no update is done. Otherwise, for $a = 1, \cdots, k-1$, $a \neq j$, the outgoing edge from β_y, from the set D_a is updated as follows.

Case 1. The vertex β_y has no outgoing edge from the set D_a. If $r^a_{z_a} > r^a_y$, then the outgoing edge from β_y from D_a is taken to be $(\beta_y \to \beta_{z_a})$.

Case 2. The vertex β_y has an outgoing edge $(\beta_y \to \beta_{y_a})$ from the set D_a. If $r^a_{z_a} > r^a_{y_a}$, then the outgoing edge from β_y from D_a is updated to be $(\beta_y \to \beta_{z_a})$.

The "shortcutting" operation from s is defined similarly, where the sets $\{e_a\}$ play the role of the sets D_a.

The example in Fig. 10 shows a shortcutting step in F_1 (the subgraph of G_B induced by $D_1 \cup \{e_1\}$).

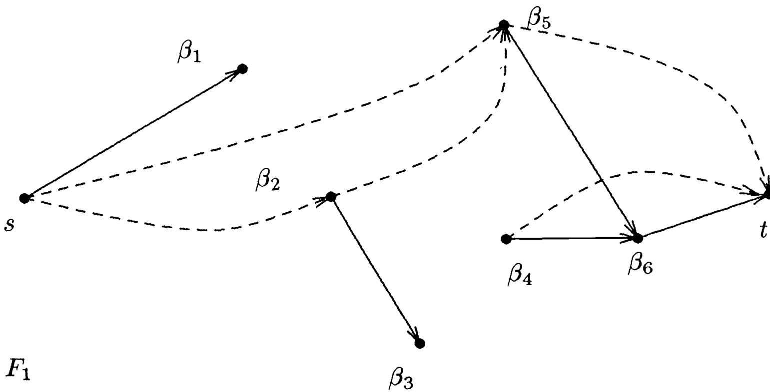


FIG. 10. *Shortcutting in* F_1.
---→ *edges added in shortcutting.*

LEMMA 4.3. *Let G'_B be the graph resulting from the "shortcutting" over the edges of $D_j \cup \{e_j\}$. Then, there exists a path from s to t in G_B, if and only if there exists a path from s to t in G'_B.*

Proof. The *if* direction. If we suppose that there is a path P'_B in G'_B, then it is easy to reconstruct a path P_B in G_B. If P'_B uses an edge $(\beta_y \to \beta_{z'})$ in G'_B (that is not an edge in G_B), then we replace it by the path from β_y to β_z and the edge $(\beta_z \to \beta_{z'})$.

The *only if* direction. For this direction we first prove the following claim.

CLAIM. *If there exists a path from s to t in G_B then there exists a path $P_B = s$, $\beta_{x_1}, \cdots, \beta_{x_a}$, t, with the following property. The edge incoming to β_{x_1} is e_{i_1}, and for each $1 < c < a$, the edge incoming to β_{x_c} is from the set D_{i_c}, where i_c is the type of the outgoing edge from β_{x_c}.*

Proof. Suppose that there exists a path $s, \beta_{y_1}, \cdots, \beta_{y_b}, t$, from s to t in G_B, that does not satisfy the property above. We show how to construct a path $P_B = s$, $\beta_{x_1}, \cdots, \beta_{x_a}$, t, with the property. This is done by replacing the edges of the original path, one by one. The edge $(s \to \beta_{y_1})$ is replaced by the edge $e_{i_1} = (s \to \beta_{x_1})$, where i_1 is the type of the outgoing edge from β_{y_1}. If β_{x_1} has an outgoing edge to t then this completes the construction. Otherwise, we add the outgoing edge from β_{x_1} from the set D_{i_2} (if such exists). Note that β_{x_1} can reach β_{y_2} by "reversing" on P_{i_1}. Thus, if β_{x_1} has no outgoing edge from D_{i_2}, then it must be the case that $r^{i_2}_{x_1} > r^{i_2}_{y_2} > l^{i_2}_{y_3}$. In this case

β_{x_2} is taken to be β_{x_1}. (That is, for the proof, we allow the repetition of vertices in P_B; the actual path is given by omitting these repetitions.) We continue in the same manner. Observe that we are guaranteed to get a path from s to t since we consistently move to a vertex whose attachment point on the path on which the next "reversal" is done is further to the right. □

We now return to the proof of Lemma 4.3. Suppose that there is a path $P_B = s, \beta_{x_1}, \cdots, \beta_{x_a}, t$, in G_B with the above property. If P_B consists only of edges from G'_B then the same path is also in G'_B. Suppose that P_B contains some edges that are not in G'_B. Let $(\beta_{x_b} \to \beta_{x_{b+1}})$ be the first such edge. We show how to construct a path $P'_B = s, \beta_{x_1}, \cdots, \beta_{x_b}, \beta_{x'_{b+1}}, \cdots, t$ in G'_B. The prefix of the path up to β_{x_b} is the same as the prefix of P_B up to β_{x_b}. If β_{x_b} has an outgoing edge to t, then we add it to P'_B to complete the path. Otherwise, we distinguish between two cases.

Case 1. The edge $(\beta_{x_b} \to \beta_{x_{b+1}})$ is not from the set D_j. The outgoing edge from β_{x_b} in P'_B is taken to be its outgoing edge $(\beta_{x_b} \to \beta_z)$ from the set $D_{i_{b+1}}$, where i_{b+1} is the *type* of the outgoing edge from $\beta_{x_{b+1}}$ in P_B. From the definition of the "shortcutting" operation and the property of P_B it follows that such an outgoing edge exists and that $r_z^{i_{b+1}} > l_{x_{b+2}}^{i_{b+1}}$.

Case 2. The edge $(\beta_{x_b} \to \beta_{x_{b+1}})$ is from the set D_j. Let $(\beta_{x_c} \to \beta_{x_{c+1}})$ be the first edge in P_B following $(\beta_{x_b} \to \beta_{x_{b+1}})$ that is *not* from the set D_j. Note that the *type* of this edge is j. In this case the vertices $\beta_{x'_{b+1}}, \cdots, \beta_{x'_c}$ are defined to be β_{x_b}. (That is, for the proof, we allow the repetition of vertices in P'_B; the actual path is given by omitting these repetitions.) There are two subcases for the outgoing edge from β_{x_c}.

Case 2.1. The vertex β_{x_c} is the root of the tree (in F_j) containing β_{x_b}. In this case the outgoing edge from $\beta_{x'_c} = \beta_{x_b}$ is taken to be its outgoing edge $(\beta_{x_b} \to \beta_z)$ from the set $D_{i_{c+1}}$ if such exists, where i_{c+1} is the *type* of the outgoing edge from $\beta_{x_{c+1}}$ in P_B. From the definition of the "shortcutting" operation it follows that if such an outgoing edge exists then $r_z^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$. Otherwise, i.e., if there is no such edge, it must be the case that $r_{x_b}^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$. In this case, $\beta_{x'_{c+1}}$ is also defined to be β_{x_b}.

Case 2.2. The vertex β_{x_c} is not the root of the tree (in F_j) containing β_{x_b}. In this case, the next vertex is determined as follows. Let β_y be the root of the tree in F_j containing β_{x_b}. We have three possibilities:

(1) If $r_{x_b}^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$ then the next vertex is also taken to be β_{x_b}.

(2) If not (1) and $r_y^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$, then let $d > c$ be the minimum index such that $r_y^{i_{d+1}} < l_{x_{d+2}}^{i_{d+1}}$. (Notice that d is always defined. The only case in which it may be undefined is when β_y has an outgoing edge to t in G_B. However, in this case β_{x_b} would have also an outgoing edge to t in G'_B; a contradiction.) The vertices $\beta_{x'_{c+1}}, \cdots, \beta_{x'_d}$ are defined to be β_{x_b}. The outgoing edge from $\beta_{x'_d} = \beta_{x_b}$ is taken to be its outgoing edge $(\beta_{x_b} \to \beta_z)$ from the set $D_{i_{d+1}}$ if such an edge exists. From the definition of the "shortcutting" operation it follows that if such an outgoing edge exists then $r_z^{i_{d+1}} > l_{x_{d+2}}^{i_{d+1}}$. Otherwise, i.e., if there is no such edge, it must be the case that $r_{x_b}^{i_{d+1}} > l_{x_{d+2}}^{i_{d+1}}$. In this case, $\beta_{x'_{d+1}}$ is also defined to be β_{x_b}.

(3) Consider the remaining possibility. Note that β_y can reach $\beta_{i_{c+1}}$ by "reversal" on P_j. Since $r_y^{i_{c+1}} \leqq l_{x_{c+2}}^{i_{c+1}}$ and $r_{x_{c+2}}^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$, β_y must have an outgoing edge $(\beta_y \to \beta_z)$ from the set $D_{i_{c+1}}$, and clearly, $r_z^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$. After the "shortcutting" operation β_{x_b} must also have an outgoing edge $(\beta_y \to \beta_z)$ from the set $D_{i_{c+1}}$, where $r_z^{i_{c+1}} > l_{x_{c+2}}^{i_{c+1}}$. This outgoing edge is taken to be the next edge in P'_B.

On obtaining the outgoing edge from β_{x_b}, we continue the above process of modifying path P_B to obtain an s-t path in G'_B. Observe that we may actually replace a vertex β_{x_c} in P_B by a vertex $\beta_{x'_c}$ in P'_B, for some $c > b$. However, we always have the property that $r_{x'_c}^{i_c} \geqq r_{x_c}^{i_c}$, for all $c > b$. In other words, we consistently move to a vertex

whose attachment point on the path on which the next "reversal" is done is further to the right. We continue modifying P_B in this manner until we reach t. □

The path from s to t in G_B is computed in $k-1$ phases. In phase j we perform the "shortcutting" operation over the edges of $D_j \cup \{e_j\}$. From the lemma above it follows that there exists a path from s to t in G_B, if and only if s is connected to t by an edge in the graph resulting after these $k-1$ phases. If such a path exists, it can be reconstructed by adding the edges deleted in the "shortcutting," starting from phase $k-1$ down to phase 1.

Next, we describe how to implement each phase in $O(\log n)$ time using $km/\log n$ processors. The implementation consists of two stages: (1) For each vertex β_y identify the root of the tree of F_j containing it. (2) Given the root, update the outgoing edges of β_y.

It is not difficult to see that Stage (2) can be implemented in constant time using km processors and hence in $O(\log n)$ time using $km/\log n$ processors. The computation of Stage (1) can be done by applying the Euler tour technique of [TV85]. However, to apply this technique we need the full adjacency list of the forest F_j, i.e., we need to compute the list of edges from the set D_j incoming to each vertex β_x. For this we use the following observation. Consider two edges of type i: $(\beta_y \to \beta_x)$ and $(\beta_z \to \beta_x)$ from the set D_j. (Since both edges are of type i we can reach β_x from both β_y and β_z, by a "reversal" over P_i.) Suppose that $r_y^i < r_z^i$. Then, all the bridges whose outgoing edge from D_j is of type i and whose rightmost attachment vertex on P_i is between r_y^i and r_z^i have an outgoing edge to β_x.

For a bridge B_x and $1 \leqq i \leqq k-1$, define the *rightmost edge* on P_i to be the edge of B_x, whose endpoint is the rightmost attachment of B_x on P_i. Consider the concatenation of the adjacency lists (in G) of all the vertices on P_i. Compact this list to include only rightmost edges of bridges whose outgoing edge from D_j is of type i. It follows from the above observation that the rightmost edges of all the bridges such that (i) their outgoing edge from D_j is of type i, and (ii) these edges point to the same vertex in G_B, are consecutive in this concatenated list. Thus, the list of incoming edges of all the vertices in G_B can be computed using the algorithms for list ranking and prefix sums in $O(\log n)$ time and $(m+n)/\log n$ processors [CV86], [AM88], [LF80].

Step 5. If there is no path from s to t in G_B, the separating set $\{w_1, \cdots, w_{k-1}\}$ can be found from the construction of Step 4 in $O(k \log n)$ time and $m/\log n$ processors. After doing the "shortcutting," assume that there is no edge from s to t in G'_B. We add the deleted vertices (that are "shortcutted" over) to G'_B, in the reverse order of deletion. At each stage of adding the vertices, we consider the set of edges which were deleted when the vertices were "shortcutted" over. For each deleted vertex, we test whether it was reachable from s when the deleted edges are added. In this way we are able to obtain the set of vertices reachable from s, and taking their rightmost attachment vertices on each of the paths yields the separating set.

If a path P_B was found, then following the proof of Theorem 4.1 we construct P_k. Recall that to make the path P_k (edge) simple we have to perform a "pruning" step. This is implemented by constructing a linked list. For each rightmost attachment vertex $r_{x_b}^i$ on P_i, define its successor succ $(r_{x_b}^i)$ to be $l_{x_d}^i$, if (i) $d = \max\{z \mid l_{x_z}^i < r_{x_b}^i\}$, and (ii) there is no $c < b$ such that $l_{x_d}^i < r_{x_c}^i$. We add condition (ii) to avoid the situation where one attachment vertex is the successor of more than one vertex. For each leftmost attachment vertex $l_{x_b}^{i_{b-1}}$, succ $(l_{x_b}^{i_{b-1}}) = r_{x_b}^{i_b}$. Since each attachment vertex has at most one successor and is a successor of at most one vertex, the resulting graph is a set of chains. A list ranking step from s yields the "pruned" path. The "pruning" is implemented

optimally in logarithmic time using the parallel prefix computations algorithm of [LF80] and the parallel list ranking algorithm of [CV86] and [AM88].

Finally, the k vertex disjoint paths can be constructed following the resulting k flow units from s to t, in $O(\log n)$ time and $n/\log n$ processors using the optimal list ranking algorithm [CV86], [AM88]. We conclude with Theorem 4.4.

THEOREM 4.4. *The described algorithm finds k vertex disjoint s-t paths in* $O(k^2 \log n)$ *time using* $kC(n, m)$ *processors.*

5. Finding edge disjoint paths. In this section we describe a parallel algorithm for finding k-edge disjoint s-t paths. The algorithms uses the same techniques as the algorithm for finding vertex disjoint paths and has the same complexity; that is, the algorithm either finds k-edge disjoint paths or a set of at most $k-1$ edges whose removal separates s and t, in $O(\log n)$ time and $C(n, m)$ processors.

The parallel algorithm is similar to the algorithm of § 4. The only difference is in the definition of the bridge graph G_B. Recall that edge disjoint s-t paths can be computed using a max-flow computation in the network $M_{s,t}$ in which each edge has unit capacity, i.e., we relax the restriction of unit vertex capacities and impose only the edge capacity constraints. Because of this relaxation the flow augmenting path P_k in $M_{s,t}$ is permitted to transfer itself from one bridge to another by using zero or more reverse edges of P_i, unlike the condition imposed earlier that required that at least one edge of P_i be transversed in the reverse direction. To capture this, the definition of the sets D_j in the bridge graph G_B is modified. Specifically, the edge $(\beta_i \to \beta_x)$ is added to the set D_j if r_x^j is to the right of r_i^j, and is the rightmost vertex on P_j among the rightmost attachments of all bridges whose leftmost attachment on some path P_y is to the left of or *the same* as r_i^y. More formally, the edge sets $D_1, \cdots, D_{k-1}$ are defined as follows. For $1 \leqq j \leqq k-1$, the edge $(\beta_i \to \beta_x) \in D_j$ if

(1) For some path $P_y (1 \leqq y \leqq k-1)$, we have $(s < l_x^y \leqq r_i^y)$.

(2) We have $(r_x^j > r_i^j)$. (This condition is considered satisfied in case B_i has no attachment vertex on P_j.)

(3) There is no β_z, such that β_z satisfies the above two conditions, and $r_z^j > r_x^j$. Note that the strict inequality in (1) was replaced by an inequality.

The correctness of the algorithm is implied by the following theorem.

THEOREM 5.1. *There are k edge disjoint paths between s and t in G if and only if there is a directed path from s to t in (the modified)* G_B.

The proof of the theorem is similar to the proof of Theorem 4.1. We show that a path in G_B corresponds to a flow augmenting path P_k in the network $M_{s,t}$. Using the flow of value k we can construct k-edge disjoint paths. If there is no path from s to t in G_B, then the edges on the paths P_i outgoing from the furthest attachment vertices of the bridges reachable from s in G_B, separate s from t.

6. Applications. In this section we describe some more applications of our parallel algorithm for finding disjoint s-t paths.

Constructing a cycle through three specified vertices.

PROBLEM. Given an undirected graph G, and three specified vertices a, b, c of G, determine whether the three vertices lie on a common simple cycle and construct such a cycle if one exists.

Below, we show how to derive an efficient logarithmic time parallel algorithm for this problem.

The parallel algorithm is a parallelization of the sequential algorithm of [LR80]. We assume that G is biconnected since a, b, and c must be in the same biconnected component if the cycle exists. The main idea of the sequential algorithm of [LR80] is

to decompose G into pieces and to look for paths which must exist in these pieces if the cycle is to exist in G. All the steps in this algorithm are easy to parallelize efficiently, except for the step that involves computing three vertex disjoint s-t paths. Using the k-disjoint paths algorithm we can compute the three paths (if such exist) and either construct the desired cycle from these paths, or conclude that such a cycle does not exist.

The algorithm can be implemented in logarithmic time using the algorithm of § 4, together with parallel logarithmic time connectivity (e.g., [CV86]), biconnectivity (e.g., [TV85]), and triconnectivity (e.g., [FRT89]) algorithms. The number of processors used in the algorithm is the same as the number of processors needed by the parallel triconnectivity algorithm. (The algorithm of [FRT89] appears to use $(n+m)\log\log n/\log n$ processors.)

The two paths problem.

PROBLEM. Given an undirected graph G, and two pairs of vertices a_1, a_2, and b_1, b_2 find two disjoint paths, one from a_1 to a_2 and one from b_1 to b_2.

Our algorithm for finding three disjoint s-t paths is used as a subroutine in the parallel algorithms of [KMV89] for the two paths problem to yield an $O(\log n)$ time, n^2 processors algorithm for solving the two paths problem on general graphs, and a logarithmic time parallel algorithm for solving the problem on planar graphs that uses the same number of processors as required for the triconnectivity algorithm.

Testing and finding subgraph homeomorphism for some fixed pattern graphs.

PROBLEM. Given a graph G, test if G has a subgraph homeomorphic to some fixed pattern graph H. If G has such a subgraph, find it.

We show how to solve this problem for the pattern graphs: K_4 and $K_{2,3}$. As a corollary, this gives an efficient algorithm to test whether a graph is outer-planar.

First, we consider the testing problem. The following lemmas are from [Asa85].

LEMMA 6.1. *For a triconnected graph H, a graph G has a subgraph homeomorphic to H if and only if there is a triconnected component of G that has a subgraph homeomorphic to H.*

LEMMA 6.2. *If a simple graph G with two or more vertices has no subgraph homeomorphic to K_4, then $m \leqq 2n-3$.*

LEMMA 6.3. *A graph G has a subgraph homeomorphic to K_4 if and only if there is a triconnected component of G with four or more vertices.*

LEMMA 6.4. *If a simple graph G with two or more vertices has no subgraph homeomorphic to $K_{2,3}$, then $m \leqq 2n-2$.*

LEMMA 6.5. *A simple graph G has a subgraph homeomorphic to $K_{2,3}$, if and only if there is a triconnected component of G satisfying one of the following:*

(i) *It has five or more vertices.*

(ii) *It is the graph K_4 with at least one virtual edge.*

(iii) *It is a triple bond of three virtual edges.*

Lemmas 6.2 and 6.3 imply a simple algorithm for testing whether a given graph G has a subgraph homeomorphic to K_4. Similarly, Lemmas 6.4 and 6.5 imply a simple algorithm for solving the same problem for $K_{2,3}$. Implementing both algorithms using a logarithmic time triconnectivity algorithm (e.g., [FRT89]) yields logarithmic time algorithms that use the same number of processors as required for the triconnectivity algorithm.

A planar graph is *outer-planar* if it can be embedded in the plane so that all its vertices lie on the same face. The following theorem is an easy corollary of Kuratowski's theorem.

THEOREM 6.6. *A graph is outer-planar if and only if it has no subgraph homeomorphic to K_4 or $K_{2,3}$.*

We conclude that we can test whether a graph is outer-planar in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm. Our algorithm is simpler and more efficient than the algorithm given in [RR89]. However, it does not yield the (outer-)planar embedding of the graph in case it is outer-planar. To obtain an outer-planar embedding we add one artificial vertex v to G, and an edge from v to every original vertex in the graph. The new graph is planar if and only if G is outer-planar, since the addition of v ensures that all the original vertices are on the same face in the obtained embedding. Using the logarithmic time algorithm in [RR89] we can obtain an outer-planar embedding for G.

Finally, we note that in a similar way we can test subgraph homeomorphism for the pattern graphs: C_4, C_5, and $K_{2,p}$, for any fixed p.

We use the k-vertex disjoint paths algorithm to obtain homeomorphs of K_4 and $K_{2,3}$.

Finding K_4 homeomorphs. Our algorthm is based on the proof of Lemma 6.3 of [Asa85]. Without loss of generality assume G has $2n$ edges (if not, choose any subgraph of G with $2n$ edges), and that G has a triconnected component G_1 with four or more vertices. Choose any vertex v of G_1 and consider the subgraph $G_1-\{v\}$ (which is biconnected). In $G_1-\{v\}$, find a cycle C of length greater than or equal to three, by obtaining a spanning tree of $G_1-\{v\}$ and adding one nontree edge. Find three vertex disjoint paths from v to three distinct vertices on the cycle C; such paths exist since G is triconnected. The three paths are found as follows. First, introduce a new vertex t, and add edges from t to all vertices on C. Then, find three vertex disjoint v-t paths. The required paths are the segments of these v-t paths up to their first point of intersection with C. It is easy to see that the graph obtained from these three disjoint paths from v to C, together with C, is homeomorphic to K_4. To get the K_4 homeomorph in G, we may need to replace virtual edges by paths in G. It follows that using our algorithm, together with parallel logarithmic time connectivity and triconnectivity algorithms, a K_4 homeomorph can be found in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm.

Finding $K_{2,3}$ homeomorphs. Our algorithm is based on the proof of Lemma 6.5 of [Asa85]. Again, we assume that G has $2n$ edges and a triconnected component G_1 satisfying one of the conditions in Lemma 6.5. We consider all the three cases which G_1 may satisfy.

Case 1. G_1 has five or more vertices: Find G', a subgraph homeomorphic to K_4. Let the vertices in G' of degree three be called v_1, v_2, v_3, v_4. If G' is exactly the graph K_4 then G_1 must have another vertex $u \notin G'$. Find three disjoint paths from u to any three vertices of the K_4. It is easy to see that a subgraph homeomorphic to $K_{2,3}$ can be extracted from these three paths and the K_4.

If G' is a subdivision of K_4, then consider a vertex u on the path $P[v_1; v_2]$. Since G_1 is triconnected, there must be a path from u to some other vertex w of G' in $G_1-\{v_1, v_2\}$. Using this path and the K_4 it becomes easy to extract the subgraph homeomorphic to $K_{2,3}$.

Case 2. G_1 is K_4 with a virtual edge: Replace the virtual edge by a path of length greater than or equal to two and obtain a subgraph homeomorphic to $K_{2,3}$.

Case 3. G_1 is a triple bond of virtual edges: Replace all three virtual edges by paths of length greater than or equal to two and obtain a subgraph homeomorphic to $K_{2,3}$.

Again, the implementation of this algorithm can be done in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm.

Acknowledgments. We are grateful to Steve Mitchell and Vijay Vazirani for useful discussions and encouragement. We are also grateful to the referees for many useful suggestions.

REFERENCES

[AM88] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in Proc. Aegean Workshop on Computing 88, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 81-90.

[Asa85] T. ASANO, *An approach to the subgraph homeomorphism problem*, Theoret. Comput. Sci., 38 (1985), pp. 249-267.

[BDD+82] M. BECKER, W. DEGENHARDT, J. DOENHARDT, S. HERTEL, G. KANINKE, W. KEBER, K. MEHLHORN, S. NÄHER, H. ROHNERT, AND T. WINTER, *A probabilistic algorithm for vertex connectivity of graphs*, Inform. Process. Lett., 15 (1982), pp. 135-136.

[Ber76] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1976.

[BM77] J. A. BONDY AND U.S.R. MURTY, *Graph Theory with Applications*, American Elsevier, New York, 1977.

[CV86] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list, tree and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 1986, pp. 478-491.

[CV88] ———, *Optimal parallel algorithms for expression tree evaluation and list ranking*, in Proc. AWOC 88, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 91-100.

[DF56] G. B. DANTZIG AND D. R. FULKERSON, *On the max-flow min-cut theorem of networks*, in Linear Inequalities and Related Systems, Annals of Math. Study, Vol. 38, Princeton University Press, Princeton, NJ, 1956, pp. 215-221.

[ET75] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507-518.

[ET76] ———, *Computing st-numbering*, Theoret. Comput. Sci., 2 (1976), pp. 339-344.

[Eve75] S. EVEN, *An algorithm for determining whether the connectivity of a graph is at least k*, SIAM J. Comput., 4 (1975), pp. 393-396.

[Eve79] ———, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[FRT89] D. FUSSEL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacements*, in Proc. 16th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 372, Springer-Verlag, Berlin, New York, 1989, pp. 379-393.

[FT88] D. FUSSELL AND R. THURIMELLA, *Separation pair detection*, in Proc. AWOC 88, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 149-159.

[Gal80] Z. GALIL, *Finding the vertex connectivity of graphs*, SIAM J. Comput., 9 (1980), pp. 197-200.

[Har69] F. HARARY, *Graph Theory*, Addison Wesley, Reading, MA, 1969.

[IR84] A. ITAI AND M. RODEH, *The multi-tree approach to reliability in distributed networks*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 1984, pp. 137-147.

[KMV89] S. KHULLER, S. G. MITCHELL, AND V. V. VAZIRANI, *Processor efficient parallel algorithms for the two disjoint paths problem and for finding a Kuratowski homeomorph*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, October 1989, pp. 300-305.

[KR87] A. KANEVSKY AND V. RAMACHANDRAN, *Improved algorithms for graph four-connectivity*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, October 1987, pp. 252-259.

[LEC67] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An algorithm for planarity testing of graphs*, in Proc. Internat. Symposium on Theory of Graphs, P. Rosenstiehl, ed., Gordon and Breach, New York, 1967, pp. 215-232.

[LF80] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831-838.

[LLW86] N. LINIAL, L. LOVÁSZ, AND A. WIGDERSON, *A physical interpretation of graph connectivity, and its algorithmic applications*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, October 1986, pp. 39-48.

[LR80] A. S. LAPAUGH AND R. L. RIVEST, *The subgraph homeomorphism problem*, J. Comput. System Sci., 27 (1980), pp. 133-149.

[Mat87] D. MATULA, *Determining edge connectivity in $O(mn)$*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, October 1987, pp. 249-251.

[MSV86] Y. MAON, B. SCHIEBER, AND U. VISHKIN, *Parallel Ear Decomposition Search* (EDS) *and st-numbering in graphs*, Theoret. Comput. Sci., 47 (1986), pp. 277-298.

[RR89] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, October 1989, pp. 282-287.

[Sey80] P. D. SEYMOUR, *Disjoint paths in graphs*, Discrete Math., 29 (1980), pp. 293-309.

[Shi80] Y. SHILOACH, *A polynomial solution to the undirected two path problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 445-456.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88-102.

[SV82] ———, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57-63.

[Tar75] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215-225.

[Thu89] R. THURIMELLA, *Techniques for the design of parallel graph algorithms*, Ph.D. thesis, Department of Computer Science, University of Texas, Austin, TX, 1989.

[TV85] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862-874.

[Wyl79] J. C. WYLLIE, *The complexity of parallel computations*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 376–394, April 1991
© 1991 Society for Industrial and Applied Mathematics
012

TIME AND MESSAGE BOUNDS FOR ELECTION IN SYNCHRONOUS AND ASYNCHRONOUS COMPLETE NETWORKS*

YEHUDA AFEK† AND ELI GAFNI‡

Abstract. This paper addresses the problem of distributively electing a leader in both synchronous and asynchronous complete networks. $O(n \log n)$ messages synchronous and asynchronous algorithms are presented. The time complexity of the synchronous algorithm is $O(\log n)$, while that of the asynchronous algorithm is $O(n)$. In the synchronous case, a lower bound of $\Omega(n \log n)$ on the message complexity is proven. It is also proven that any message-optimal synchronous algorithm requires $\Omega(\log n)$ time. In proving these bounds, the type of operations performed by nodes are not restricted. The bounds thus apply to general algorithms and not just to comparison-based algorithms.

Key words. distributed algorithms, leader election algorithms, complete networks, time-message complexities trade-off

AMS(MOS) subject classifications. 68M10, 68Q, 68R, 94C15

1. Introduction. In the *election* problem, a single node, called the leader, is to be selected from a set of nodes which initially differ only in their identifiers (ids), with no node being aware of any other id. An arbitrary subset of nodes wakes up spontaneously at arbitrary times and starts the algorithm by sending messages over the network. When the message exchange terminates, a leader is distinguished from all other nodes.

This paper specializes in the problem of electing a leader in a complete network. In such a network, each pair of nodes is connected by a bidirectional communication link. Before the algorithm starts, no node has any information on any of the other nodes. Hence, the incident links of a node, on which no message was sent or received, are indistinguishable. We consider both synchronous and asynchronous modes of communication.

When designing an algorithm, one has to take into consideration the class of inputs for the algorithm. The smaller the class, the more efficient an algorithm might be found since special features of the class may be exploited. Thus, for electing a leader in an arbitrary topology asynchronous network, a $\Theta(m + n \log n)$ bound on the message complexity was proved, [Bur80], [FL87], [GHS83], [Gal77], [PKR82] where n and m are the total number of nodes and links in the network. $\Omega(m)$ is clearly a lower bound for an arbitrary network, since no algorithm may terminate before sending at least one message over each link, as an untraversed link could be the only link connecting two parts of the network, each holding a separate election. While the $\Omega(n \log n)$ part of the lower bound holds for networks as sparse as a ring [Bur80], [FL87], [FL85], Korach,

* Received by the editors November 11, 1987; accepted for publication (in revised form) June 4, 1990. A preliminary version of §3 was presented at the Twenty-Second Annual Allerton Conference on Communication, Control, and Computing, Allerton, Illinois, October 3-5, 1984. A preliminary version of §4 was presented at the Fourth Annual Association of Computing Machinery Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 1985.

† Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel; and AT&T Bell Laboratories, Murray Hill, New Jersey 07974. The research of this author was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense under contract MDA 903-82-C-0064.

‡ Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel; and University of California, Los Angeles, California 90024. The research of this author was supported by National Science Foundation Presidential Young Investigator Award under grant DCR84-51396 and matching funds from the XEROX Corporation under grant W881111.

Moran, and Zaks [KMZ84] noted that the $\Omega(m)$ lower bound *does not* hold for the smaller class of complete networks, where an election algorithm may terminate after one node has communicated with all its neighbors. Subsequently, they proved a lower bound of $\Omega(n \log n)$ messages for asynchronous complete networks, and presented an algorithm that requires $5n \log n + O(n)$ messages and $O(n \log n)$ time.

For synchronous networks, it was shown by Frederickson and Lynch [FL87], [FL85] that one should distinguish between two types of algorithms: *general*, in which nodes may perform any computation on the values of their ids, and *comparison*, in which the values of ids can only be used for comparison with each other. They addressed the problem of election in a synchronous ring, for which they presented a general algorithm with $O(n)$ messages, and proved a lower bound of $\Omega(n \log n)$ messages for comparison algorithms, thus showing that general algorithms are strictly more powerful than comparison algorithms in a ring. This difference stems from the capability of general algorithms in synchronous networks to delay messages and processors as a function of the value of their id's.

In this paper we prove that the message complexity of any election algorithm, comparison or general, in a complete synchronous or asynchronous network is $\Theta(n \log n)$, thus proving that general algorithms are not more powerful than comparison algorithms for the problem of election in complete networks. The difference between synchronous rings and synchronous complete networks stems from the fact that in a ring all nodes can be distributively awakened with n messages, whereas in the complete network the awakening problem is as hard as the election problem, hence requiring $\Omega(n \log n)$ messages. Moreover, we prove that $\Omega(n \log n)$ is a lower bound on the message complexity of any algorithm whose execution requires the sending of messages, at least over the links of a connected spanning subnetwork. If all the nodes of a complete network could be awakened with n messages, then a general algorithm could take advantage of the synchronous mode of communication to elect a leader in a linear number of messages by using the principles suggested in [FL87], [Gaf85], [Vit84].

Consider the following, straightforward synchronous election algorithm in complete networks. Every initiator starts the algorithm by sending messages, containing its id, to all its neighbors. All the nodes then elect the highest id initiator as the leader. The time complexity of this algorithm is two time units, and its worst-case message complexity is $\Theta(n^2)$. In §3.1 we slow down this simple algorithm and design an $O(\log n)$ time, message-optimal synchronous algorithm. This is done by carefully selecting a dynamic rate at which initiators send messages to their neighbors. Furthermore, by varying the dynamic rate at which initiators send messages to their neighbors, we are able to trade message complexity for a better time complexity. Hence, we present a continuum of $2 \log_c n$-time, $2c \cdot n \cdot \log_c n$-messages, $2 \leq c \leq n$, synchronous algorithms, closing the gap between the trivial $O(1)$-time, $O(n^2)$-messages algorithm and the $O(\log n)$-time, $O(n \log n)$-messages algorithm.

When applying the dynamic rate technique in the asynchronous model, we obtain a $5 \cdot n \log n$-messages but still linear time algorithm. The asynchronous algorithm is an improvement over the algorithm of [KMZ84], which takes $O(n \log n)$ time and $5n \log n$ messages.

In an effort to reduce the message complexity of the asynchronous algorithm to $2n \log n$, we present a sequence of three asynchronous algorithms (A, B, and C). The first two algorithms present a trade-off between time and message complexities. Algorithm A has $O(n \cdot \log n)$-time complexity and $2 \cdot n \cdot \log n$-message complexity. Algorithm

B (which was also derived independently in [Hum84]) has $O(n)$-time complexity but $2.773 \cdot n \cdot \log n$-message complexity. Analyzing the communication and time complexities of the two algorithms, we derive a third algorithm, Algorithm C, whose time complexity is $O(n)$ and communication complexity is $2n \log n$, an improvement on the $O(n \log n)$-time and $2n \log n$-message algorithm of Peterson [Pet84].

Following the $\Omega(n \log n)$ lower bound on the message complexity and the continuum of synchronous algorithms, we prove an $\Omega(\log n)$ lower bound on the time complexity of any message-optimal election algorithm in synchronous complete networks.

Finally, we prove a tight lower bound on the continuum of trade-off between the message and time complexities of the synchronous algorithm. Specifically, we show that if an algorithm, whether comparison or general, elects a leader in at most $\frac{1}{2} \cdot \log_c n$ rounds, then its message complexity is at least $((c-1)/2)n \cdot \log_c n$. Thus showing that on the one hand, any message-optimal algorithm has an $\Omega(\log n)$-time lower bound, and on the other hand, any $O(1)$-time algorithm has an $\Omega(n^2)$-message lower bound.

The models used in this paper are described in §2. In §3 we present the various algorithms. In §4 we present the lower bounds for the synchronous case, thus proving the optimality of our algorithms. Throughout the paper, unless otherwise specified, we use log to denote $\log_2$.

2. Models and assumptions. In this section we present the synchronous and asynchronous complete network models which follow the traditional message-passing model [Bur80], [LF81].

In a *complete network* of n nodes (processors), every node is connected by $n - 1$ bidirectional communication links to all the other nodes. Nodes have unique ids, but no node knows the id of any other node. Each link incident to a node has a unique representation in that node. All messages received at a node are stamped with the identification of the link through which they arrived. By the number of its outgoing links every node knows n before the algorithm starts. However, *all the links incident to a given node on which no message was sent or received are indistinguishable to this node.*

The nodes of the network are initially asleep, but an arbitrary subset wakes up at arbitrary times and starts the distributed election algorithm. Each node in this subset is called *initiator.* Sleeping nodes may be awakened by receiving a message from the algorithm. When the message exchange terminates, exactly one node emerges as the network leader.

Considered here are two modes of communication, *synchronous* and *asynchronous.* In the *synchronous* mode of communication, a global clock is connected to all the nodes in the network. The time interval between two consecutive pulses of the clock is a *round.* At the beginning of each round, each node decides, according to its state, what messages to send and on which links to send them. Each node then receives any messages sent to it in this round and uses all the received messages and its state to decide on its next state. Initiators start the distributed algorithm by entering an initial state and then waiting for the beginning of the next round.

In the *asynchronous* mode of communication, there is no global clock and messages incur arbitrary but finite delay. Messages from all input links are transferred into a central queue. The processor receiving the messages processes them one at a time in the order that they arrive at the central queue. The processing time of a message is negligible in comparison to its communication delay.

Throughout the paper, the *message complexity* (*communication cost*) of an algo-

rithm is the worst-case total number of messages sent during the algorithm, where a message contains at most $O(\log n)$ bits. The time complexity is the worst-case total number of time units from the first to the last message transmission due to the algorithm, where a time unit in the synchronous case is one round. In the asynchronous case, and only for the purpose of time complexity analysis, one time unit is an assumed upper bound on message transmission delay. In arguing the time complexity, we shall allow a message to traverse a link in any fraction of the time unit.

3. Complete networks election algorithms. In this section we derive a distributed algorithm for election in synchronous complete networks and three algorithms for election in asynchronous complete networks. The message complexity of all the algorithms is $O(n \log n)$.

The underlying mechanism for all the algorithms is similar. Each algorithm is initiated by any subset of nodes, each of which is a *candidate* for leadership. Each candidate tries to capture all other nodes by sending messages on all the links incident to it. The algorithm terminates when a candidate has succeeded in capturing all its neighbors. This candidate becomes the leader. To guarantee that only one node is elected, all candidates but one are *killed.*

All candidates use a variable called *level* to estimate the number of nodes they have already captured. The level variable is used by candidates to contest each other. Captured nodes also have a level variable, which tracks the highest level candidate they have observed. That candidate is their *owner.* All level variables are initialized to 0.

A candidate that sees a node with a larger level than its own is eliminated from candidacy. However, if the candidate's level is larger or equal, the node's level is replaced by the candidate's level. The candidate may then claim the node and try to eliminate the previous owner of the node.

The algorithms differ mainly in three parts: (1) The way candidates determine their level, (2) The way candidates capture their neighbors, and (3) The rule candidates use to eliminate each other. In the synchronous algorithm, the level of a candidate is the number of rounds since it started the algorithm, and a candidate at level $2i$ tries to capture 2^i new nodes simultaneously (in one round). Unlike the synchronous algorithm, in the asynchronous algorithms candidates capture one neighbor at a time. In the first asynchronous algorithm, Algorithm A, the level of a candidate is the number of other candidates it has killed. In Algorithm B, the level of a candidate is the number of nodes it has already captured (following [Gal77].) Algorithm A achieves a better message complexity while Algorithm B achieves a better time complexity. In Algorithm C, candidates use a combination of the above two level functions to attain the time complexity of B and the message complexity of A.

To simplify the algorithms, every initiator node spawns two processes, the *candidate* process and the *ordinary* process. The two processes are connected to each other by a bidirectional logical link which behaves like a physical link. A node awakened by receiving a message from the algorithm spawns only an ordinary process. Candidate processes communicate only with ordinary processes and vice versa. Thus, the communication topology can be viewed as a complete bipartite graph, on one side the candidate processes and on the other side n ordinary processes. Henceforth, the term *candidate* will be applied interchangeably to both the process and its initiating node. All messages received by a node are tagged according to the type of their sending process. Messages received from candidate processes are forwarded to the ordinary process. Messages received from ordinary processes are forwarded to the

candidate process.

3.1. The synchronous algorithm. In this section we present a $2 \cdot \log n$-rounds, $3n \log n$-messages synchronous algorithm. In the next section we will show that this algorithm is message optimal and is as fast as a message-optimal algorithm can be.

The algorithm is given in Fig. 1. At every candidate the algorithm proceeds in levels. Every live candidate at level $2i$, $i \geq 0$, tries to capture 2^i new ordinary processes by sending them messages containing its level and id. If in level $2i+1$ the candidate receives acknowledgments from all the ordinary processes it tries to capture, then it proceeds as a candidate to the next level. Otherwise, the process (and hence the node owning it) is eliminated from candidacy.

The level of a candidate is incremented by one every round. Every ordinary process has *level* and *owner_id* variables that contain the level and id of the highest-level candidate the process has received a message from (level ties are resolved by selecting the highest id). In every round, every ordinary process first increases its level by one, to reflect the owner's actual level, and then inspects the newly received messages to update its level and owner_id if necessary. If an update occurs, the ordinary process acknowledges its new owner.

In the algorithm description, E is the set of edges incident to a candidate process. Every candidate maintains a list of edges, called *untraversed*, which it has not yet traversed in any direction.

3.1.1. Time and message complexities. Let p be the largest id of a candidate from the set of oldest candidates (i.e., whose level is the largest). We observe the following three facts.

FACT 1. *The level of every ordinary node strictly increases from one round to the next.*

FACT 2. *At most $n/2^{i-1}$ candidates reach level $2i$, $1 \leq i \leq \log n$.*

FACT 3. *$2 \log n$ rounds after it has started the algorithm, candidate p has captured all the nodes and is elected as the network leader.*

Fact 1 follows immediately from the algorithm for ordinary node processes. Fact 3 holds because all the messages of p get acknowledged, and once a node has acknowledged p, it will not acknowledge any other message. Fact 2 follows from Fact 1 and the observation that every ordinary node acknowledges at most one message in which the level is i, $0 \leq i \leq \log n$, i.e., the sets of 2^{i-1} nodes that are captured by each candidate that has reached level $2i$ are disjoint.

Following Fact 3, the time complexity of the algorithm is $2 \log n$. Since every node sends at most one acknowledgment to a candidate in level $2i$, the total number of acknowledgments is $n \log n$, each of length $O(1)$ bits. Due to Fact 2, the total number of candidate messages is $\sum_{i=1}^{\log n} (n/2^{i-1})2^i = 2n \log n$, each message containing $\log n + \log\log n$ bits. The total communication complexity is thus $3 \cdot n \log n$ messages.

A continuum of algorithms can be devised to close the gap between the trivial $O(1)$-time, $O(n^2)$-messages algorithm and the $O(\log n)$-time, $3n \log n$-messages algorithm. Each algorithm in the continuum is the same as the above, except that a candidate in level i tries to capture c^i neighbors, $2 \leq c \leq n$. The time complexity of the algorithm is $2 \log_c n$, and its message complexity is $2c \cdot n \cdot \log_c n$, thus proving that the lower bounds that will be presented in §4.3 (Theorem 4.5) are tight.

3.1.2. Asynchronizing the synchronous algorithm. To maintain the $O(n \log n)$ message complexity in the asynchronous communication mode, we are

```
Candidate program:
    untraversed ← E
    level ← -1 ;
    Each round do:
        level ← level + 1 ;
        if level is even
        then
            if untraversed is empty
            then
                Elected, Stop
            else
                K ← Minimum ( 2^(level/2),|untraversed|) ;
                Send (level, id) over K links from untraversed, and
                remove these links from untraversed;
        else                                                /* level is odd */
            Receive all acknowledgment type messages
            if received less than K acknowledgments
            then
                Stop                              /* Not a candidate anymore */
    End each round.

Ordinary program:
    l* ← nil ;
    level ← -1 ;
    owner_id ← id ;
    Each round do:
        Send an acknowledgment over l* ;
        level ← level + 1 ;
        Receive all candidate messages (level,id) over link l;
        Let (level*, id*) be the lexicographically largest
        (level, id) candidate message, and
        l* the link over which it arrived ;
        if (level*, id*) > (level, owner_id)
        then
            (level, owner_id ) ← (level*, id*) ;
        else
            l* ← nil ;
    End each round.
```

FIG. 1. *The Synchronous Algorithm.*

forced to increase the time complexity to $O(n)$. The increase in the time complexity seems unavoidable and it remains an open question whether a sublinear-time, message-optimal asynchronous algorithm exists.

There are two basic differences between the asynchronous and synchronous modes of communication. First, in the asynchronous mode there is no global clock, and second, messages incur an arbitrary but finite delay. The arbitrary delay of messages (and not the absence of the clock) is the source of the increase in the time complexity of the algorithm. Essentially, the synchronous algorithm would perform as well without a global clock, if all messages sent by a single node incur exactly the same delay.

To see that a straightforward application of the synchronous algorithm in the asynchronous model will not work consider the following situation: There are two competing candidates, C_1 and C_2; each has already successfully captured one node, and both proceed to capture the nodes v and u at the same time. A message of C_1 was the first to arrive at v which is then captured by C_1, while a message of C_2 was first to arrive at u. Following the rules of the synchronous algorithm, v positively acknowledges only C_1 and u positively acknowledges only C_2. Thus, both candidates are killed, since none had all of its messages positively acknowledged, and the algorithm deadlocks. In the following sections we present algorithms which overcome this and similar problems in the asynchronous case.

3.2. Three asynchronous algorithms. Our aim in this subsection is to derive a $2 \cdot n \log n + O(n)$-messages, linear-time asynchronous algorithm. To this end we present a sequence of three asynchronous algorithms (A, B, and C), each devised to circumvent the problems of the previous one, so that Algorithm C achieves the desired complexity.

Unlike the synchronous algorithm, a candidate process in the asynchronous algorithms tries to capture all the other nodes by successfully traversing its incident links one at a time.

3.2.1. Algorithm A. Level: In this algorithm, the level of a candidate is the *total number of other candidates that it has killed.*

Capturing and elimination rule: To capture node v, the level of a candidate must be strictly larger than that of v, in which case the candidate captures v without killing the previous owner of v. When two live candidates in the same level encounter each other, the higher id candidate kills the other and increases its level by one.

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

If $Level(P) < Level(v)$, P is killed.
If $Level(P) > Level(v)$, v is captured by P, and v gets P's level.
If $Level(P) = Level(v)$, P is sent to Q.
Upon arriving at Q:
If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.
If Q has already been killed, P is killed too.
If $(Level(P), id(P)) > (Level(Q), id(Q))$, **then** (1) Q is killed, (2) P increases its level by one, and (3) P captures v.

Details: To keep track of its owning candidate, every captured node has two link pointers, *father* and *potential_father.* The father pointer points to the link through which the node was most recently captured, and the potential_father pointer points to the link through which a candidate which tries to claim the node from its father, has arrived.

As in the synchronous algorithm every initiator spawns two independent processes, *candidate* and *ordinary.* The two processes are connected by a bidirectional logical link that behaves like a physical link.

A formal description of Algorithm A is given in Fig. 2. When a candidate arrives at node v whose level is the same as its own, and the id of v's father, Q, is smaller, it becomes v's potential_father. The potential_father is then sent to Q in an attempt to kill it. If another candidate at the same level with even higher id arrives at v before the potential_father returns from Q, then this other candidate is killed. If the potential_father survives at Q it first increments its level by one, then returns to v and captures it, and only then, returns to its initiating node. However, if the potential_father finds that Q is already killed, it eliminates itself as well (since Q is killed, there must exist a higher level candidate in the network).

Analysis: Since at most half of the candidates at level k go up to level $k+1$, the maximum level achievable during the algorithm is $\log n$. Clearly, every time a node is recaptured its level is increased by at least one. Hence, the total number of capture messages possible is at most $n \cdot \log n$. Each capture uses 2 messages, which sums up to a total of $2 \cdot n \cdot \log n$ messages. The extra messages sent by candidates which go over father links to other candidates is at most $2 \cdot n$, since each such traversal results in the elimination of one live candidate. Thus, the message complexity of the algorithm is $2 \cdot n \cdot \log n \ + \ 2 \cdot n$ messages, each of length $\log n \ + \ \log\log n$ bits.

The time complexity of the algorithm is $O(n \cdot \log n)$ by the following scenario, in which $n/2$ of the nodes are captured serially $\log(n/2)$ times. The algorithm is started by node v_0, which captures $n/2$ nodes in level 0. Then, a new node, v_1, spontaneously starts the algorithm, kills v_0, increases its level to 1, and recaptures the same $n/2$ nodes. After v_1 has captured the $n/2$ nodes, two new nodes spontaneously start the algorithm, try to kill each other, and the one which survives, v_2, reaches level 1. Node v_2 then kills v_1 and recaptures the $n/2$ nodes at level 2. The scenario continues until the entire network has been captured by $v_{\log n/2}$, which is elected as a leader.

The basic reason for the high time complexity of the algorithm is that the level of a candidate is not a function of the number of nodes it has already captured. A candidate which spent a lot of work (and time) accumulating nodes might be killed by a candidate which did not spend nearly as much. In the next algorithm we will employ a linear level function, suggested by Gallager in a similar context [Gal77], thus reducing the time complexity to $O(n)$.

3.2.2. Algorithm B. Level: In this algorithm, the level of a candidate is the *number of nodes it has already captured.*

Capturing and elimination rule: To capture node v, (1) the (level, id) of a candidate must be lexicographically larger than the (level, id) of the previous owner of v, and (2) the previous owner must be killed.

When candidate P arrives at node v, which is currently owned by candidate Q, the following rule is used:

If $(Level(P), id(P)) \ < \ (Level(v), id(Q))$, P is killed.

If $(Level(P), id(P)) \ > \ (Level(v), id(Q))$, (1) v gets P's level, and (2) P is sent to Q.

When P arrives at Q:

If $(Level(P), id(P)) \ < \ (Level(Q), id(Q))$, P is killed.

If Q has been killed already **then** P returns to v and retries to capture it.

```
Initially:
    level ← size ← 0 ; owner_id ← potential_id ← 0 ; /* size used in Algorithm C only*/
    untraversed ← E ; father ← potential_father ← nil ; killed ← false ;

Candidate (id) :
while (untraversed ≠ φ) do;
    l ← any(untraversed) ; send(level, id) on l ;
R:  receive(level′, id′) over l′ ;
    if (id′ = id)AND not killed then                          /*successful capturing.*/
               level ← level′ ; untraversed ← untraversed - l ;
    else-if (level′, id′ < level, id) OR killed               /* lexicographically*/
    then Discard the message, goto R ;
    else send(level′, id′) over l′; killed ← true ; goto R ;
end while;
if not killed then announce(Elected);

Ordinary:
    level ← -1 ;
    while (not terminated) do;
        receive(level′, id′) over l′ ;
        case level′ of :
        (1) level′ < level : Discard message ;
        (2) level′ > level:                                   /* Replace the father */
                      father ← l′ ; level ← level′ ; owner_id ← id′ ;
                      potential_id ← 0; potential_father ← nil ;
                      send(level′, id′) over the father link ;
        (3) level′ = level :
                      if (id′ < owner_id) then Discard message;
                      else-if (id′ = potential_id) then
                          father ← potential_father ;
                          level′ ← level′ + 1 ;
                          owner_id ← id′ ; potential_id ← 0 ;
                          potential_father ← nil ;
                          send(level′,id′) over the father link ;
                      else-if there is already a potential_father then Discard message ;
                      else                        /* there is no potential_father */
                          potential_id ← id′ ; potential_father ← l′ ;
                          send(level′, id′) over the father link ;
        end case ;
    end while ;
```

FIG. 2. *Algorithm* A.

If $(Level(P), id(P)) > (Level(Q), id(Q))$, **then** (1) Q is killed, and (2) P returns to v and retries to capture it.

Upon returning to its initiating node from a successful capturing, P increases its level by one.

Details: A candidate C that arrives at an already captured node v whose level is smaller than its own, replaces v's level with its own and becomes v's $potential_father$. C is then sent to the father candidate of v. If C survives at v's father, and meanwhile no other candidate replaced C as the potential_father of v, then C becomes v's father. If v has not yet been captured, the potential_father automatically becomes the father of v. A formal description of Algorithm B is given in Fig. 3.

Analysis: The algorithm is deadlock-free since candidates never wait for each other, and the (*level*, *id*) pair is lexicographically increasing along any chain of candidates that kill each other.

The time complexity of the algorithm is $O(n)$ since candidates never wait for each other and a candidate that has done more work is never killed by a candidate that has done less work. Thus, each killed candidate spent, in the worst case, less time than the one killing it.

To prove that the communication complexity of the algorithm is $O(n \log n)$ we use a Lemma which was introduced in [Gal77].

LEMMA 3.1. *For any given k, the number of candidates that own n/k or more nodes is at most k.*

Proof. Let C_1 and C_2 be any two candidates which owned n/k nodes at some point in time. We shall show that each of C_1 and C_2 must have owned disjoint sets of at least n/k nodes each. If they never tried to claim a node from each other, we are done. The first time that C_1 (without loss of generality) tries to claim a node, say v, from C_2, either one of them dies, or C_2 has already been killed. If C_1, without loss of generality, caused the death of C_2, then clearly it must have owned at least n/k nodes disjoint from C_2, at the time of killing. If C_2 is already dead, C_1 must still own at least n/k nodes in order to claim v to itself. □

COROLLARY 3.2. *The largest candidate to be killed by another candidate owns at most $n/2$ nodes; the next largest owns at most $n/3$ nodes, etc.*

LEMMA 3.3. *The message complexity of Algorithm* B *is* $4 \cdot n \cdot \ln n$ $(= 2.773 \cdot n \cdot \log_2 n)$ *messages.*

Proof. Since in capturing one node a candidate makes at most 4 hops, a candidate which owned k nodes incurs at most $4 \cdot k$ messages. By Corollary 3.2, the total cost is then bounded by $4 \cdot n \cdot \sum_{i=1}^{n} (1/i)$ messages. Note that each message of the algorithm contains at most $2 \cdot \log n$ bits. □

The number of candidates at a particular level was constrained by the disjointness property. Hence, a candidate which captures many nodes from another candidate tries to eliminate that other candidate as many times as the number of nodes it captures from it. This gives rise to the factor 4 in the message complexity.

In Algorithm A the amount of work a candidate spent was not factorized into the level function and thus the time complexity was $O(n \log n)$. Although in Algorithm B the time complexity was reduced to $O(n)$, it has the problem that candidates could be re-eliminated many times, thus increasing the message complexity. In the next algorithm we eliminate both problems by employing a level function which is a combination of the two.

3.2.3. Algorithm C. Here we make two modifications to Algorithm A in order to achieve a linear-time complexity with no increase in the communication cost. First,

```
Initially
level ← owner_id ← 0 ;
untraversed ← E ; father  ← nil  ; killed ← false ;

Candidate ( id ) :
while ( untraversed ≠ φ ) do;
      l ← any( untraversed ) ;
      send(level, id) on l ;
R:    receive(level', id') over l' ;
      if (id' = id)AND not killed then                    /* successful capturing.*/
                  level ← level + 1 ;
                  untraversed ← untraversed - l ;
      else                    /*another candidate tries to eliminate candidate id */
                  if (level', id' < level, id)             /* lexicographically*/
                  then Discard the message, goto R ;
                  else                                 /* Candidate id is eliminated */
                      send(level', id') over l';
                      killed ← true ;
                      goto R ;
end while;
if not killed announce(Elected);

Ordinary:
for_ever do;
      receive(level', id') over l' ;
      case level', id' of :
      (1) level' ,id' < level, owner_id:
                Discard message ;
      (2) level', id' > level, owner_id :
                potential_father ← l' ;
                level ← level' ;
                owner_id ← id' ;
                if father = nil then father ← potential_father ;
                send(level', id') over the father link ;
      (3) level', id' = level, owner_id:
                father ← potential_father ;
                send(level', id') over the father link ;
      end case ;
end for_ever ;
```

FIG. 3. *Algorithm* B.

we incorporate an estimate of the amount of work spent by each candidate into the level function of Algorithm A. Second, we enable candidates with a high level ($> \log n$) to capture many nodes in parallel (in one time unit). We start describing the algorithm with the first modification. The second modification will be introduced during the performance analysis.

Level: In this algorithm the level of a candidate is increased according to two rules. 1) The rule of Algorithm A, and 2) After each capture the candidate increases its level to be at least log (*total number of nodes captured*), i.e., after returning from a successful capture the level is set to max (log (*# nodes captured*), *present level*).

Capturing and Elimination Rule: This rule is the same as in Algorithm A.

Details: The formal description of the algorithm is similar to that of Algorithm A. A variable, called "*size*," is used to count the number of nodes captured by a candidate. The only change is to replace the first "then" clause, within the "while" loop of a candidate program to:

then
 $size \leftarrow size + 1$;
 $level \leftarrow \max(level', \log size)$;
 untraversed ← untraversed - 1 ;

Analysis: To analyze its performances we will first show the following lemma.

LEMMA 3.4. *The maximum* level *reachable during any execution of Algorithm* C *is* $\log n + \log\log n + 1$.

Proof. Let N_i be the total number of candidates that reach level i during the execution of the algorithm. Consider the maximum number of candidates which could possibly pass from level $i-1$ to level i. There are two ways in which a candidate can go from level $i-1$ to level i. First, by capturing 2^{i-1} nodes at level $i-1$ for $i \le \log n$, and second, by killing another candidate which is at level $i-1$. We note that N_i is maximized if as many candidates as possible pass from level $i-1$ to level i by capturing other nodes (i.e., $n/2^{i-1}$ candidates) and the rest of the candidates (i.e., $N_{i-1} - (n/2^{i-1})$) kill each other in pairs. Hence,

$$N_i \le \frac{(N_{i-1} - (n/2^{i-1}))}{2} + \frac{n}{2^{i-1}}. \tag{1}$$

Solving (1) for N_i we get:

$$N_i \le \frac{n}{2^i} \cdot (i+1). \tag{2}$$

Substituting $N_i = 1$ in (2) and solving for i gives us the maximum level, which is $\log n + \log\log n + 1$. □

Using the same argument as in Algorithm A we find that the message complexity of Algorithm C is $2 \cdot n \cdot (\log n + \log\log n + 2)$ messages, each of length $\log n + \log(\log n + \log\log n)$ bits.

With the above modification the time complexity of Algorithm A is reduced to $O(n \cdot \log\log n)$. This is due to the fact that if the highest level in the network is i, it must increase within time $\min(2^i, n)$. In order to further reduce the time complexity to $O(n)$, processes at levels higher than $\log n$ will try to capture $n/\log n$ nodes in parallel. Thus a candidate which has reached level $\log n$ will send messages over $n/\log n$ untraversed links incident to it. Each of these messages carries the (level, id) of the candidate. When a message arrives at an adjacent node the node compares

its level to that of the message. If the message level is higher, the node replaces its (level, id) with the message, thus making the candidate the new father of the node. The node then sends the candidate an acknowledgment of successful capture. If the message level is smaller, it returns no message. Finally, if the message level is the same as that of the node but the message id is higher, a notification to that effect is sent back to the candidate.

The candidate waits for all the $n/\log n$ acknowledgments. If all the acknowledgments indicate a successful capture, the candidate proceeds to the next $n/\log n$ untraversed incident links. If, on the other hand, some of the acknowledgments indicate that they have encountered the same level, one of the links is arbitrarily chosen and a process that behaves as in Algorithm A is sent along that link. If the process returns, the candidate increases its level and proceeds to the next $n/\log n$ untraversed links (links on which no successful capture was reported are not considered traversed).

To analyze the algorithm with this modification we make two observations: First, the maximum attainable level in the algorithm is still bounded by $\log n + \log\log n + 1$. Second, by substituting $i = \log n$ in (2), we find that the maximum number of candidates which reach level $\log n$ is $\log n$.

The last modification has increased the communication complexity of the algorithm by at most $O(n)$ messages. Each node is still captured at most $\log n + \log\log n + 1$ times; however, the death of a candidate at level greater than $\log n$ might be associated with at most $2 \cdot n/\log n$ messages. Since there are at most $\log n$ such candidates the increase due to killings is bounded by $O(n)$.

To show that the time complexity of the algorithm is $O(n)$ we arrange the candidates in a rooted tree. Each level of the tree corresponds to the candidates that have reached that level in the algorithm, i.e., the vertices at level i in the tree correspond to the candidates that have reached level i in the algorithm. The parent of a candidate at level i in the tree is either the candidate that caused the death of the given candidate or the same candidate at the next level. The time delay of the algorithm is the sum of the delays incurred by candidates along the path from the first initiator (candidate) to wake up, at level 0, to the root.

To evaluate this time delay we note that no candidate that either survives or is killed at level i spends more than 2^{i-1} time units in level i, $i \leq \log n$. In level i, $\log n < i \leq \log n + \log\log n$, no candidate spends more than $\log n$ time units, since it captures nodes at a rate of $n/\log n$ per time unit. Hence, the total time delay of the algorithm is bounded by $\sum_{i=1}^{\log n} 2^i + \log n \cdot \log\log n = n + \log n \cdot \log\log n$. Note that we scale a time unit to be the maximum delay it takes to capture one node, which is a constant.

In the above calculation we did not include the actual time it takes candidates to kill each other. Since there are at most n candidates and no candidate tries to kill a dead one (unlike Algorithm B), this delay is also bounded by $O(n)$.

4. Lower bounds for election in complete networks. Two algorithms for election in synchronous complete networks were discussed in the previous sections. The first is an $O(n^2)$-message $O(1)$-time algorithm and the second is an $O(n \log n)$-messages $O(\log n)$-time algorithm. The two algorithms raise two questions: (1) Is $\Omega(n \log n)$ also the lower bound on the message complexity of election in synchronous complete networks, and (2) If $\Omega(n \log n)$ is the message complexity lower bound, then how fast can a message-optimal algorithm be, i.e., is there an $O(n \log n)$-messages-$O(1)$-time algorithm, or is $\Omega(\log n)$ the lower bound on the time complexity of any message-optimal synchronous algorithm.

In this section we answer these questions by proving first, that $\Omega(n \log n)$ is a lower bound on the worst-case message complexity of a synchronous algorithm, and second, by proving that $\Omega(\log n)$ is a lower bound on the time complexity of any message-optimal synchronous algorithm. In proving these bounds we do not restrict the type of operations performed by the nodes. The bounds thus apply to general algorithms and not just to comparison-based algorithms. This proves that the synchronous algorithm of §2 is optimal.

Furthermore, in §2 we have presented a continuum of $(2c/\log c) \cdot n \log n$-messages $2 \cdot \log_c n$-time synchronous algorithms where $c = 2, \cdots, n$. In Theorem 4.5 each algorithm in the continuum is shown to be optimal by proving that if an algorithm (whether comparison or general) elects a leader in at most $\frac{1}{2} \cdot \log_c n$ rounds, then its message complexity is at least $(c-1)/(2 \cdot \log c) \cdot n \log n$.

To show the lower bounds, a scenario in which any synchronous (and hence also asynchronous) algorithm transmits at least $n/2 \cdot \log n$ messages, is constructed using an adversary argument. A similar argument is then used to show that the delay of any message-optimal algorithm is at least $\Omega(\log n)$ rounds.

4.1. Definitions and assumptions. Consider an arbitrary election algorithm on the synchronous model defined above. An *event* is the sending of a message over a previously unused link (two messages sent in the same round in opposite directions over a previously unused link are considered two separate events). With each event we associate a link (s, d), where s is the source node and d is the destination node of the corresponding message. With each round i of the algorithm we associate a set of events, R_i. A sequence $E = (R_0, R_1, \cdots)$ is called an *execution.* An *execution-prefix* E_j is a prefix, $(R_0, R_1, \cdots, R_j)$, of an execution E. With each run of the algorithm we associate an execution, called a *legal-execution*, that includes all events which occurred in the run, arranged in order of the corresponding rounds. Henceforth, any mention of a message refers to an event.

With each execution-prefix E_j we associate a graph G_j, whose nodes are the nodes of the network and whose links are the links associated with the events in E_j. A *cluster* in an execution-prefix E_j is any maximally connected component of G_j. The "*degree*" of node v in an execution-prefix E_j is the degree of v in G_j. The *potential-degree* of node v in an execution-prefix E_j is the degree of v in E_j plus the number of times that v is a source node of an event in R_{j+1}. The *potential-degree* of a set of nodes is the maximum potential-degree among its nodes.

For the purpose of proving the lower bounds we introduce a slightly different model than the standard synchronous model, called the *stopping-model.* The stopping-model allows us to withhold the clock pulse, at the beginning of round j, from any set of clusters, C, in E_{j-1}, given that no node in C is expected to receive a message in round j from a node not in C. The nodes in C are then said to be *frozen* in round j. Therefore, a frozen node in a round neither sends nor receives any message in that round; nor does it change its state. The stopping-model will be used to prevent large differences in the clusters' growth rates.

The stopping-model essentially allows us to select any cluster in an execution-prefix E_j and send it back in time, i.e., the nodes of the cluster are awakened some number of rounds later than they were in E_j.

A *stopping-execution* is an execution which corresponds to a run of the algorithm in the stopping-model. A stopping-execution is called a k *stopping-execution* if the cumulative number of pulses withheld over all frozen sets of clusters throughout the run is k. Obviously, a 0-stopping-execution is a legal-execution.

LEMMA 4.1. *For any k stopping-execution E, there exists a $k-1$ stopping-execution E' which contains exactly the same events as E does.*

Proof. Let l be the minimum index of a round in which any set of clusters is frozen, and let C be a set of clusters which is frozen in l. An execution-prefix E' which satisfies the lemma can be obtained from E by shifting all events which occurred before round l and involve nodes in C, one round forward. This affects neither any event in later rounds nor any event which involves nodes not in C. This is because neither E nor E' contains an event connecting a node in C with a node not in C in any round R_j, $j \leq l$, and because R_{l+1} in E is identical to R_{l+1} in E'. Notice that in E' the nodes in C are awakened one round later than in E. □

COROLLARY 4.2. *For any stopping-execution there exists a legal-execution which contains the same events.*

In the next two sections we will prove the lower bounds on the stopping model. Using Corollary 4.2, these bounds apply also to the nonstopping model. In our proofs we do not restrict the type of operations performed by the nodes, hence proving the bounds for general algorithms.

4.2. A lower bound on message complexity. At the end of any election algorithm all nodes know who the leader is, hence any such algorithm has to send messages along the links of a spanning subnetwork. In other words, by the end of the algorithm the whole network is contained in one cluster. Thus, no cluster in the algorithm can defer indefinitely the sending of messages to nodes not in the cluster, as the rest of the network might not wake up spontaneously.

In the following proof of the lower bound we will use an adversary argument to construct a stopping-execution (and thus a legal-execution) which contains at least $\frac{1}{2}n \log n$ events. In the beginning of each round, the adversary determines first which clusters to freeze, and then which links to use for the new events generated by the unfrozen nodes. The first feature is used to delay the formation of larger clusters until later rounds in the run, thus avoiding large differences in the clusters' growth rates; the second feature is used to send as many messages as possible within one cluster, and thus again minimizing the cluster's growth rate. The second feature is possible since links incident to a given node on which *no* message was sent or received are indistinguishable to this node.

In the proof of Theorem 4.3 an $(n/2) \cdot \log n$ event execution-prefix is constructed formally. We will first demonstrate the construction through an example.

EXAMPLE 1. *Assume n is a power of 2. A stopping execution-prefix with at least $(n/2) \cdot \log n$ events is constructed in $\log n$ phases, each lasting few rounds. In phase j, $j = 0, \cdots, \log n$, the nodes are partitioned into sets of size 2^j, such that every subset contains two subsets of phase $j-1$. The adversary continues the execution of (does not freeze) any subset in phase j, whose potential-degree is less than $2^j - 1$, all other subsets are frozen. The destination of events generated by nodes whose potential-degree is less than 2^j is chosen by the adversary to be within the same subset. Thus, no cluster with more than 2^j nodes is created in phase j and every cluster is contained in one subset. Each phase is ended when all the nodes are frozen, i.e., when in every subset there is a node which tries to send a message to a node not in the subset.*

Table 1 gives the potential degrees of the subsets by the end of each phase. In phase 0 of the example all nodes are awakened. Nodes 1, 2, 3, 5, 7, $\cdots$, 13, 15, 16 have potential-degree 1, node 4 2, node 6 5, and node 14 9. If any of these nodes started with potential-degree 0, all the nodes with potential-degree greater than 0 are frozen until eventually all will have potential-degree at least 1. Thus in phase 0 no

TABLE 1
Potential-degrees of nodes in the example.

<table>
<tr><th>Node id
Phase</th><th>1
•</th><th>2
•</th><th>3
•</th><th>4
•</th><th>5
•</th><th>6
•</th><th>7
•</th><th>8
•</th><th>9
•</th><th>10
•</th><th>11
•</th><th>12
•</th><th>13
•</th><th>14
•</th><th>15
•</th><th>16
•</th></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>2</td><td>1</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>9</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="2">5</td><td colspan="2">2</td><td colspan="2">2</td><td colspan="2">4</td><td colspan="2">9</td><td colspan="2">2</td></tr>
<tr><td>2</td><td colspan="4">4</td><td colspan="4">5</td><td colspan="4">4</td><td colspan="4">9</td></tr>
<tr><td>3</td><td colspan="8">8</td><td colspan="8">9</td></tr>
</table>

event was generated. In phase 1, *node* 3 *is frozen because node* 4 *has potential-degree* 2. *Similarly node* 5 *is frozen because of node* 6, *and* 13 *because of* 14. *By the end of the first phase the constructed execution-prefix,* E_{i_1}, *contains the following events:* $\{(1,2)(2,1)(7,8)(8,7)(9,10)\ (10,9)(11,12)(12,11)(15,16)(16,15)\}$. *By the end of the second phase,* E_{i_2} *contains, at least, the following additional events:* $\{(3,4)\ (4,3)(4,2)\ (2,3)\}$. E_{i_3}, *by the end of the third phase, contains, at least, the following additional events:* $\{(4,1)(4,5)(6,5)(6,7)(6,8)(6,4)(6,3)(8,5)\}$. *In the first round of the fourth phase the following events will be added:* $\{(6,2)(6,1)(6,9)(11,10)(11,9)(11,13)(13,14)\ (14,9)(14,10)(14,11)\ (14,12)(14,13)(14,15)\ (14,16)(14,8)(14,7)(15,14)\}$. *After these events the whole network is contained in one cluster and a good election algorithm could terminate without generating any more events.*

Clearly, by the end of phase j there are at least $n/2^j$ nodes with potential-degree 2^j. Thus every phase eventually contributes at least $n/2$ new events, summing up to $(n/2) \cdot \log n$ events. If n is not a power of 2, then $n/2 < 2^m < n < 2^{m+1}$ and the scenario will be constructed over 2^m nodes; the rest of the nodes will be awakened only by the end of the last phase. In the next theorem we prove the lower bound formally.

THEOREM 4.3. *A stopping-execution of an election algorithm in a synchronous complete network of n nodes contains at least $(n/2) \cdot \log n$ events, in the worst case.*

COROLLARY 4.4. *The message complexity of any election algorithm in a synchronous complete network of n nodes is at least $(n/2) \cdot \log n$.*

Proof of Theorem 4.3. Assume without loss of generality that $n = 2^q$. We define a sequence of partitions $(P_0, \cdots, P_q)$ of the nodes such that each subset in partition P_0 contains one node, and each subset in P_j contains two subsets from P_{j-1}, $1 \leq j < q$. Hence, each subset in P_j contains 2^j nodes.

We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, \cdots, E_{i_q})$, $i_0 = 0$, each being a prefix of the next. E_{i_0} is an empty execution-prefix in which all nodes have been awakened and the potential-degree of each node is at least 1. This is done by withholding the clock pulse from any node whose potential-degree is at least 1 until there is no node with potential-degree 0. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset in P_j and (2) The potential-degree in E_{i_j} of every subset in P_j is at least 2^j. Obviously, E_{i_0} satisfies these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, we describe how the adversary constructs E_{i_j}, $j = 1, \cdots, q-1$. In each round of phase j, we freeze all the subsets in P_j whose potential-degree greater than or equal to 2^j. When all subsets are frozen, phase j is completed. The source and destination nodes of any message sent in this phase are both in the same subset in P_j. This is always possible, since every node that has a potential-degree greater than or equal to 2^j is frozen. Clearly, E_{i_j} satisfies the inductive assumptions. In the qth phase no freezing takes place. After that phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

Clearly, there are at least $n/2^j$ nodes whose degree at the end of the algorithm is at least 2^j, for $j = 0, \cdots, q-1$. Thus, the total number of events is at least $(n/2) \cdot \log n$. □

Given that the message complexity of any election algorithm on a synchronous complete network is $\Omega(n \log n)$, the question arises of how fast a message-optimal algorithm can be. In the next section we prove that the time complexity of any message-optimal algorithm is $\Omega(\log n)$.

4.3. A lower bound on time complexity. In this section we will extend the techniques of the previous section to prove that the shorter the length of the execution, the larger the lower bound on the number of events it must contain.

THEOREM 4.5. *Any stopping-execution of an election algorithm in a synchronous complete network of n nodes which terminates in less than* $\frac{1}{2} \cdot \log_c n$ *rounds, contains at least* $(c-1)/(2 \cdot \log c) \cdot n \log n$ *events.*

COROLLARY 4.6. *The time complexity of any message-optimal election algorithm in a synchronous complete network of n nodes is* $\Omega(\log n)$ *rounds.*

Proof of Theorem 4.5. Consider an election algorithm whose time complexity is at most $\frac{1}{2} \cdot \log_c n$. Assume without loss of generality that $n = c^q$. A construction similar to the proof of Theorem 4.3 will be used here. We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, \cdots, E_{i_q})$, $i_0 = 0$, each being a prefix of the next, and a sequence of partitions $(P_0, \cdots, P_q)$, the subset of each partition containing c subsets of the previous partition. Each subset of P_0 contains one node, thus each subset of P_j contains c^j nodes. E_{i_0} is an empty execution-prefix in which all nodes are awakened spontaneously. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset of P_j, and (2) The potential-degree in E_{i_j} of every subset in P_j is at least c^j. Obviously, E_{i_0} and P_0 satisfy these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, the adversary constructs E_{i_j} by first constructing partition P_j, and then E_{i_j}. Let $(S_1, \cdots, S_k)$, $k = n/c^{j-1}$ be the subsets of P_{j-1} indexed in nondecreasing order of their potential-degrees in $E_{i_{j-1}}$. Then the ith subset of P_j is defined as the union of $S_{(i-1)c+1}, \cdots, S_{ic}$, $i = 1, \cdots, n/c^j$. This implies that with the exception of at most one subset in P_j, called the *boundary* subset, each of the subsets in P_j consists of subsets in P_{j-1}, all of which either have potential-degree above or equal to c^j or below it.

In each round of phase j, $j = 1, \cdots, q-1$, we freeze all the subsets in P_j whose potential-degree is greater than or equal to c^j. When all subsets are frozen phase j is complete. The destinations for messages to be sent by node v are selected from the subsets which included v in partitions $P_0, \cdots, P_j$, in that order of priority. This is always possible since every node that has a potential-degree greater than or equal to c^j is frozen. Clearly, E_{i_j} and P_j satisfy the inductive assumptions. After the qth phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

We now show that every node is the destination of at least $\frac{1}{2} \cdot (c-1) \log_c n$ events in E_{i_q}, hence Theorem 4.3.

As the time complexity is at most $q/2$, every node must have been frozen in all the rounds of at least $q/2$ phases. Otherwise, the legal-execution corresponding to E_{i_q} would contain more than $q/2$ rounds, contradicting the assumption on the time complexity. If node v is frozen in all the rounds of phase j, and it is not in a boundary subset, it will later receive one message from every subset of P_{j-1} which does not contain v and is with v in a subset of P_j. Thus, for each phase in which v is frozen in all its rounds, v is the destination of $c-1$ events. The total number of events in E_{i_q}

TABLE 2
Potential degrees in the time lower bound proof.

Phase									
Phase 1	1 2 3	4 5 6	7 8 9	10 11 12	13 14 15	16 17 18	19 20 21	22 23 24	25 26 27
	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●
	5	9	5	4	4	20	4	10	4
Phase 2	10 11 12	13 14 15	19 20 21	25 26 27	1 2 3	7 8 9	4 5 6	22 23 24	16 17 18
	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●
	4	4	4	4	5	5	9	10	20
	ACTIVE			ACTIVE			FROZEN		
	19			18			20		

is thus at least $(c-1) \cdot (q/2) \cdot n - U$, where U is the number of events that should be discounted due to the boundary subsets. Due to the boundary subset in phase j, at most $c^{j-1} \cdot (c-1)$ events should be discounted. Hence, U equals at most $n \cdot c$ and the total number of events is thus at least $(c-1)/(2 \cdot \log c) \cdot n \log n - n \cdot c$. □

EXAMPLE 2. *In Table 2 we show how E_{i_2} is constructed from E_{i_1} assuming $n = 27$ and $c = 3$. Subsets $\{4, 5, 6\}$, $\{22, 23, 24\}$, and $\{16, 17, 18\}$ are frozen during phase 2, because node 5 has potential-degree 9, node 23 10, and node 17 20. These subsets will become active again in phase 3, and then each node in them will be the destination of at least two new events, e.g., nodes 4, 5, and 6 each will receive a message from node 23 and 17. Nodes 22, 23, and 24 each will receive a message from nodes 5 and 17, and nodes 16, 17, and 18 will receive a message from nodes 5 and 23.*

5. Conclusions. The effect of synchronous and asynchronous communication on the problem of distributively electing a leader in a complete network was examined. On the one hand, it was proved that the message complexity is not affected by the choice of the communication mode. In both modes of communication, the message complexity was shown to be $\Theta(n \log n)$. With synchronous communication, the time complexity of message-optimal algorithms was proved to be $\Theta(\log n)$, whereas with asynchronous communication, only an $O(n)$ upper bound on the time complexity was obtained. The lower bound on the time complexity in asynchronous communication remains an open question and is the subject of the following conjecture.

CONJECTURE. *The time complexity of any message-optimal asynchronous election algorithm on a complete network is $\Omega(n)$.*

The implication of the conjecture is that synchronous communication is faster by a factor of $n/\log n$ than asynchronous communication. An analogous result was obtained in [AFL83], where a particular synchronous system of parallel processors was proved to be faster by a factor of $\log n$ than the corresponding asynchronous system.

Three asynchronous election algorithms (A, B, and C) were presented. The simplicity of the complete network topology, and hence, of termination detection, enabled us to concentrate on the synchronization among contending candidates. With each of the three algorithms we can associate an analogous algorithm for arbitrary topology networks, which uses the corresponding method to synchronize different initiations of the algorithm but a different method to traverse the network (i.e., to detect termination). The analogy to Algorithm B was given in [Gal77]. In [GHS83] the same level function as in Algorithm A was used, however, in [GHS83] candidates merge their "territories" (rather than kill each other) when they meet. In [Gaf85] the time complexity of [GHS83] is improved by replacing its level function with that of Algorithm

C. Awerbuch [Awe87] used the ideas of Algorithm C and [Gaf85] to further improve the time complexity of [GHS83] to $O(n)$. Each of the methods can be applied to other classes of networks. For example, applications of method A in different classes of topologies are discussed in [KKM85]. As in Algorithm A, the time complexity of the algorithms in [KKM85] is $O(n \log n)$, whereas similar applications, but of methods B or C, improve the time complexity of [KKM85] while maintaining the message optimality.

REFERENCES

[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.

[Awe87] B. AWERBUCH, *Linear time distributed algorithms for minimum spanning trees, leader election, counting and related problems,* in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, May 1987, pp. 230–240.

[Bur80] J. E. BURNS, *A formal model for message passing systems,* Tech. Report, Computer Science Department, Indiana University, Bloomington, IN, May 1980.

[FL85] G. N. FREDERICKSON AND N. A. LYNCH, *A general lower bound for electing a leader in a ring,* Tech. Report CSD-TR-512, Computer Science Department, Purdue University, West Lafayette, IN, March 1985.

[FL87] ———, *Electing a leader in a synchronous ring* J. Assoc. Comput. Mach., 34 (1987), pp. 98–115.

[Gaf85] E. GAFNI, *Improvements in the time complexity of two message-optimal election algorithms,* in Proc. Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 1985.

[Gal77] R. G. GALLAGER, *Finding a leader in a network with $O(e) + O(n \log n)$ 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.

[Hum84] P. A. HUMBLET, *Selecting a leader in a clique in $O(n \log n)$ messages,* in Proc. 23rd IEEE Conference on Decision and Control, Las Vegas, NV, December 1984, pp. 1139–1140.

[KKM85] E. KORACH, S. KUTTEN, AND S. MORAN, *A modular technique for the design of efficient distributed leader finding algorithms,* in Proc. Fourth ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 1985.

[KMZ84] E. KORACH, S. MORAN, AND S. ZAKS, *Tight lower and upper bounds for some distributed algorithms for a complete network of processors,* in Proc. Third ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 1984, pp. 199–207.

[LF81] N. A. LYNCH AND M. J. FISCHER, *On describing the behavior and implementation of distributed systems,* Theoret. Comput. Sci., 13(1981), pp. 17–43.

[Pet84] G. L. PETERSON, *Efficient algorithms for elections in meshes and complete networks,* Tech. Report, Department of Computer Science, University of Rochester, Rochester, NY, August 1984.

[PKR82] J. PACHL, E. KORACH, AND D. ROTEM. *A technique for proving lower bounds for distributed maximum-finding algorithms,* in Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, 1982, pp. 378–382.

[Vit84] P. VITANYI, *Distributed election in an Archimedean ring of processors,* in Proc. 16th ACM Symposium on Theory of Computing, Washington, DC, 1984, pp. 542–547.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 395–403, April 1991
©1991 Society for Industrial and Applied Mathematics
013

GOOD AND BAD RADII OF CONVEX POLYGONS*

PETER GRITZMANN†, LAURENT HABSIEGER‡, AND VICTOR KLEE§

Abstract. The "radii" considered here are the inradius ρ, the circumradius R, the diameter δ, and the width Δ. The convex polygons in question have their vertices at points of the integer lattice in $\mathbb{R}^2$, and their radii are measured with respect to an ℓ^p norm. Computation of these radii for convex polygons (and of their higher-dimensional analogues for convex polytopes) is of interest in connection with a number of applications, and may be regarded as a basic problem in computational geometry. The terms *good radius* and *bad radius* refer to the existence or nonexistence of a *rationalizing polynomial*—a nonconstant rational polynomial q such that $q(\varphi(C))$ is rational whenever C is a convex lattice polygon and φ is the radius function in question. When a radius is good, the polynomial is a tool for implicit computation of the radius in the binary model of computation; otherwise it seems to be necessary to resort to approximation. It is proved here that all four radii are good when $p \in \{1, \infty\}$, while δ is good when p is an integer and Δ is good when $p/(p-1)$ is an integer. Thus δ and Δ are both good when $p = 2$, and it turns out that R is also good in this case. However, the main results are that r is bad when $p = 2$ and R is bad for each integer $p \geq 3$.

Key words. convex lattice polygon, polarity, width, diameter, inradius, circumradius, algebraic number, rationalizing polynomial, implicit computation

AMS(MOS) subject classifications. 52A10, 52A25, 52A40, 90C05, 12D05, 68A20, 68U05, 11J72

Introduction. Throughout this paper, $\mathbb{M}_p$ denotes the Minkowski plane formed by equipping the Cartesian plane $\mathbb{R}^2$ with the norm $\| \ \|_p$. This norm is defined for $1 \leq p < \infty$ by setting $\|(\xi, \eta)\|_p = (|\xi|^p + |\eta|^p)^{1/p}$ and for $p = \infty$ by setting $\|(\xi, \eta)\|_\infty = \max\{|\xi|, |\eta|\}$. For each $p \in [1, \infty]$ the number $\overline{p}$ is defined by the condition that $1/p + 1/\overline{p} = 1$, so that $\mathbb{M}_{\overline{p}}$ may be identified with the conjugate space of $\mathbb{M}_p$.

We are interested in the problem of computing four important measures of a convex polygon C whose vertices lie in the integer lattice L in $\mathbb{M}_p$.

With respect to a given norm $\| \ \|$ the *inradius* $\rho(C)$ is the radius of a largest disk contained in C and the *circumradius* $R(C)$ is the radius of a smallest disk containing C. (The *disk* of *radius* r and*center* x is $\{y : \|x - y\| \leq r\}$.) The *diameter* $\delta(C)$ is the maximum of the distances between points of C, and the *width* $\Delta(C)$ is the minimum of the distances between pairs of parallel lines that support C. It is obvious that $2\rho \leq \Delta \leq \delta \leq 2R$.

To set the stage, we note that the computation of δ seems to be trivial, for $\delta(C)$ is the maximum of $\|v_i - v_j\|$ over all pairs (v_i, v_j) of vertices of C. However, if we are concerned with the binary (Turing machine) model of computation, then irrational

*Received by the editors June 12, 1989; accepted for publication (in revised form) June 13, 1990. Most of this paper was written while the three authors were visiting the Institute for Mathematics and Its Applications, 206 Church Street S.E., Minneapolis, Minnesota 55455.

†Institut für Mathematik, Universität Augsburg, Universitätsstr. 8, D-8900 Augsburg, Federal Republic of Germany. The research of this author was supported in part by the Alexander-von-Humboldt Foundation and by the Deutsche Forschungsgemeinschaft.

‡Département de Mathématique, Université du Québec, Montréal, Québec H3C 3P8, Canada.

§Department of Mathematics, University of Washington, Seattle, Washington 98195. The research of this author was supported in part by the National Science Foundation.

numbers are not directly accessible and hence it is usually impossible to compute $\delta(C)$ explicitly even when p is an integer ≥ 2. We must instead be content to compute $\delta(C)^p$, or in other words to determine $q(\delta(C))$, where q is the rationalizing polynomial given by $q(\xi) = \xi^p$. The situation is similar with the number $\Delta(C)$, which may be irrational and hence not directly accessible; but when $\overline{p}$ is an integer the number $\Delta(C)^{\overline{p}}$ is rational (see §1). In each of these cases, an irrational number is "computed" implicitly by explicit evaluation of an appropriate *rationalizing polynomial*. And since the polynomial is monotone in these two cases, comparing the diameters or the widths of two polygons is an easy task. A principal purpose of this paper is to show (by means of the results stated in the abstract) that rationalizing polynomials do not always exist, even for such simple functions as the inradius and the circumradius, even when attention is restricted to very simple polygons, and even when monotonicity of the polynomial is not required. In fact, §2 exhibits a family $\{T_\alpha : \alpha \in \mathbb{N}\}$ of isosceles triangles with vertices in L and a family $\{W_\alpha : \alpha \in \mathbb{N}\}$ of right triangles with vertices in L such that each nonconstant rational polynomial q has the following two properties: (i) for each fixed integer $p \geq 3$, the number $q(R(T_\alpha))$ is irrational for infinitely many α; (ii) for $p = 2$, the number $q(\rho(W_\alpha))$ is irrational for all but finitely many α. Thus in a sense these circumradii and inradii are computationally intractable. The intractability persists in higher dimensions.

When there is no rationalizing polynomial, one is naturally led to wonder how efficiently the radii in question can be approximated. That question is considered (in n dimensions) in another paper [5], which also discusses applications of radii and studies the complexity of radius computations in the cases where rationalizing polynomials are known to exist. The present paper focuses mainly on the case of two dimensions.

As general references, we mention [2] for convex bodies, [3] and [6] for computational geometry.

1. Some good radii. For convex lattice polygons in $\mathbb{M}_p$, the good radii that we know about are those mentioned in the following theorem.

THEOREM 1.1 (GOOD RADII OF CONVEX LATTICE POLYGONS). *The following numbers are rational for each convex lattice polygon C in $\mathbb{M}_p$:*

$$\begin{array}{ll} \delta(C)^p & \textit{when } p \in \mathbb{N} \cup \{\infty\}; \\ \Delta(C)^{\overline{p}} & \textit{when } \overline{p} \in \mathbb{N} \cup \{\infty\}; \\ R(C)^p & \textit{when } p \in \{1, 2, \infty\}; \\ \rho(C)^{\overline{p}} & \textit{when } \overline{p} \in \{1, \infty\}. \end{array}$$

(*In these statements, μ^∞ is defined to mean μ. And of course, $p \in \{1, \infty\}$ if and only if $\overline{p} \in \{1, \infty\}$.*)

Proof. The assertion is obvious for the diameter function δ. For the width function Δ, recall that $\Delta(C)$ is the minimum of the distances between pairs of parallel supporting lines of C. Each line of a minimizing pair includes at least a vertex of C, and it will be proved that there is a minimizing pair (S, S') for which $S' \cap C$ is an edge of C. When C is a lattice polygon, the line S' is determined by two lattice points and hence has an equation of the form $\alpha\xi + \beta\eta = \gamma$, where α, β, and γ are integers. Let $\nu = \|(\alpha, \beta)\|_{\overline{p}}$, the norm of the linear functional φ on $\mathbb{M}_p$ given by $\varphi(\xi, \eta) = \alpha\xi + \beta\eta$, and let v be a vertex of C that lies in S. Then $\nu^{\overline{p}}$ is an integer and the distance $\Delta(C)$ between S and S' is equal to the fraction $|\varphi(v) - \gamma|/\nu$. Since the numerator of this fraction is an integer, the number $\Delta(C)^{\overline{p}}$ is rational.

When the unit disk D is smooth, *every* minimizing pair (S, S') is of the described sort. However, the preceding paragraph requires only the *existence* of such a mini-

mizing pair, and that is assured in an arbitrary Minkowski plane. Indeed, consider an arbitrary minimizing pair (J, J') and suppose that each of the lines intersects C in a single point — say $J \cap C = \{v\}$ and $J' \cap C = \{v'\}$. If the distance between J and J' is Δ, then there is a translate G of the disk $\frac{1}{2}\Delta D$ such that $J \cap C = \{v\}$ and G is supported by both J and J'. Since C is a polygon, the parallel lines J and J' can be rotated (about v and v', respectively) through an angle θ in either direction to new positions J_θ and J'_θ so as to retain (for a while) the property of being supporting lines of C. The line J_θ intersects G. If the direction of rotation is properly chosen, then J'_θ also intersects G and hence the distance between J_θ and J'_θ is at most Δ. Thus the rotation can be continued until one of the lines hits a vertex of C in addition to the one about which it is being rotated. That completes the discussion of Δ.

The circumradius $R(C)$ is the radius of a smallest disk containing C. For each choice of three vertices u, v, and w of C, let R_{uvw} denote the circumradius of the triangle determined by these three vertices. By a routine application of Helly's theorem on the intersection of convex sets, $R(C)$ is the minimum of the numbers R_{uvw}. From this observation it follows when $p = 2$ that C's circumcenter is equidistant from some three vertices of C or is the midpoint of a segment joining two vertices of C. When C is a lattice polygon, this implies readily that both coordinates of the circumcenter are rational and hence $R(C)^2$ is rational.

Now suppose that $p \in \{1, \infty\}$, and for each point (ξ, η) in the plane, let

$$\begin{aligned} \phi(\xi,\eta) &= \xi \quad \text{and} \quad \psi(\xi,\eta) = \eta \qquad && \text{when } p = \infty, \\ \phi(\xi,\eta) &= \xi + \eta \quad \text{and} \quad \psi(\xi,\eta) = \xi - \eta \qquad && \text{when } p = 1. \end{aligned}$$

Then in each case, ϕ and ψ are linear functionals of norm 1 and

$$\|(\xi,\eta)\| = \max\{|\phi(\xi,\eta)|, |\psi(\xi,\eta)|\}.$$

Using this fact, it is not hard to see that for each polygon C and for $p \in \{1, \infty\}$,

$$R(C) = \max\left\{\frac{\max\phi(C) - \min\phi(C)}{2}, \frac{\max\psi(C) - \min\psi(C)}{2}\right\}.$$

This number is of course rational when C is a lattice polygon.

We turn finally to the inradius $\rho(C)$, with $p = 1$ or $p = \infty$. Let (ξ_1, η_1), (ξ_2, η_2), (ξ_3, η_3), (ξ_4, η_4) be the vertices of the unit disk, and let $H_1, \cdots, H_k$ be closed halfplanes whose intersection is the convex polygon C. Then the following conditions are satisfied whenever ρ is the radius of a disk that is contained in C and has center (α, β) :

$$(\alpha,\beta) + \rho(\xi_i,\eta_i) \in H_j \qquad (1 \le i \le 4, 1 \le j \le k).$$

Each of these conditions expresses a linear inequality constraint in the three real variables ρ, α, and β, and the inradius $\rho(C)$ is the maximum of ρ subject to these $4k$ constraints. When C is a lattice polygon, all of the coefficients in the constraints are rational and from this it follows that $\rho(C)$ is rational. □

The "goodness" results of Theorem 1.1 should be compared with our later "badness" results Theorems 2.3 and 2.5, and with Problem 3.1.

Theorem 1.1 is a special case of a theorem proved in the paper [4] for arbitrary finite-dimensional ℓ^p spaces. The proof of Theorem 1.1 is included here because it is simpler and more instructive than the general proof, and because we want the present paper to be self-contained for the two-dimensional case. The four "radii" ρ, R, δ, and Δ are special instances of functions studied in higher-dimensional spaces in [4], [5]. For each convex body C in an n-dimensional Minkowski space $\mathbb{M}$, and

each integer j with $1 \le j \le n$, the *inner j-radius* $r_j(C)$ is the radius of a largest j-dimensional ball contained in C and the *outer j-radius* $R_j(C)$ measures how well C can be approximated, in a minimax sense, by a j-flat in $\mathbb{M}$. If B denotes the unit ball of $\mathbb{M}$, then $r_j(C)$ is the maximum of the numbers r such that $(c + rB) \cap F \subset C$ for some point c of C and j-flat F containing c, and $R_j(C)$ is the minimum of the numbers s such that $C \subset G + sB$ for some $(n-j)$-flat G in $\mathbb{M}$. When $n = 2$, our ρ, R, δ, and Δ are, respectively $r_2, R_2, 2r_1$, and $2R_1$.

Let $\mathbb{R}^n_p$ denote the n-dimensional Minkowski space that results from $\mathbb{R}^n$ by applying the norm $\| \ \|_p$. (Thus $\mathbb{R}^2_p$ is the $\mathbb{M}_p$ of the Introduction.) The purpose of [5] is to study the complexity of computing the various radii of an n-dimensional convex lattice polytope P in $\mathbb{R}^n_p$, and in that connection it is important to know about the algebraic tractability of these radii. It is proved in [4] that *if P is an n-dimensional convex lattice polytope in $\mathbb{R}^n$, then the following numbers must be rational: $r_1(P)^p$ when $p \in \mathbb{N} \cup \{\infty\}$; $R_1(P)^{\overline{p}}$ when $\overline{p} \in \mathbb{N} \cup \{\infty\}$; $R_n(P)^p$ when $p \in \{1, 2, \infty\}$; $r_n(P)^{\overline{p}}$ when $\overline{p} \in \{1, \infty\}$.*

2. Some bad radii. The present section contains our main results, which establish the algebraic intractability of the circumradii or inradii of certain lattice triangles. (See [1] for a similar study of some other geometric quantities.)

We begin with two lemmas, assuming henceforth that p is an integer ≥ 2.

LEMMA 2.1 (AN EQUATION SATISFIED BY CERTAIN CIRCUMRADII). *For each $\alpha \in \mathbb{N}$, let T_α denote the triangle in the plane $\mathbb{M}_p$ whose vertices are $(-1, 0), (0, \alpha)$, and $(1, 0)$, and let R_α and c_α denote the circumradius and circumcenter of T_α. Then the first coordinate of c_α is 0, and*

$$R^p_\alpha - (\alpha - R_\alpha)^p = 1.$$

Proof. With $c_\alpha = (\gamma_1, \gamma_2)$, the fact that $\gamma_1 = 0$ follows readily from the axial symmetry common to T_α and the unit disk of the space. It is also clear that

$$1 + \gamma_2^p = R^p_\alpha \quad \text{and} \quad (\alpha - \gamma_2)^p = R^p_\alpha.$$

Since $R_\alpha > 0$, it follows from the second equation that $\gamma_2 = \alpha - R_\alpha$, and then substitution into the first equation yields the stated conclusion. $\square$

LEMMA 2.2 (ON IRREDUCIBILITY). *Suppose that p is odd and α is even, or that p is even and α is a prime that does not divide p. Then the polynomial*

$$q_0(\xi) = \xi^p - (\alpha - \xi)^p - 1$$

is irreducible over the rational field $\mathbb{Q}$ (and hence the R_α of (2.1) is an algebraic number of degree p).

Proof. Expanding the polynomial in terms of powers of ξ, we get

$$q_0(\xi) = \begin{cases} 2\xi^p - p\alpha\xi^{p-1} + \cdots + p\alpha^{p-1}\xi - \alpha^p - 1 & \text{for } p \text{ odd;} \\ p\alpha\xi^{p-1} - \cdots + p\alpha^{p-1}\xi - \alpha^p - 1 & \text{for } p \text{ even.} \end{cases}$$

Now we apply Eisenstein's criterion for irreducibility, which says that the polynomial $q(\xi) = \alpha_n\xi^n + \cdots + \alpha_1\xi + \alpha_0$ is irreducible over $\mathbb{Q}$ if there exists a prime number π such that π divides $\alpha_0, \cdots, \alpha_{n-1}$ but not α_n, and π^2 does not divide α_0. Observe that the polynomial $\xi^n q(\xi^{-1})$ is reducible if and only if $q(\xi)$ is reducible. Thus the criterion also applies with the order of coefficients reversed.

The lemma's assertion follows from an application of Eisenstein's criterion with $\pi = 2$ in the first case and $\pi = \alpha$ in the second case. $\square$

For an easy example of the situation in Lemma 2.2, note that when $p = 3$ the real root of q_0 is

$$R_\alpha = \left(\frac{1}{4} + \frac{1}{8}\left(\alpha^6 + 4\right)^{1/2}\right)^{1/3} + \left(\frac{1}{4} - \frac{1}{8}\left(\alpha^6 + 4\right)^{1/2}\right)^{1/3} + \frac{1}{2}\alpha.$$

THEOREM 2.3 (COMPUTATIONAL INTRACTABILITY OF CIRCUMRADII). *Let p be a fixed integer greater than 2, let the numbers R_α be as in Lemma 2.1, and suppose that q is a nonconstant rational polynomial. Then $q(R_\alpha)$ is irrational for all but finitely many even $\alpha \in \mathbb{N}$ when p is odd, and for all but finitely many prime α when p is even.*

Proof. For the case in which p is odd (and ≥ 3) and α is even, we begin with the division of q by the polynomial q_0 of Lemma 2.2. Let $s = s(\xi, \alpha)$ and $t = t(\xi, \alpha)$ be rational polynomials such that

$$q(\xi) = s(\xi, \alpha) q_0(\xi, \alpha) + t(\xi, \alpha),$$

where the degree $\deg_\xi t$ of t with respect to ξ is at most $p - 1$. Then we may write

$$t(\xi, \alpha) = \eta_0(\alpha) + \eta_1(\alpha)\xi + \cdots + \eta_{p-1}(\alpha)\xi^{p-1},$$

where the η_i are rational polynomials. For the particular choice of $\xi = R_\alpha$, this yields

$$q(R_\alpha) = t(R_\alpha, \alpha) = \eta_0(\alpha) + \eta_1(\alpha) R_\alpha + \cdots + \eta_{p-1}(\alpha) R_\alpha^{p-1},$$

and since R_α is an algebraic number of degree p, this implies that

$$\eta_1(\alpha) = \cdots = \eta_{p-1}(\alpha) = 0$$

for each even α such that $q(R_\alpha)$ is rational. If this happens for infinitely many even α, then the polynomials η_i are all identically zero and the polynomial $t(\xi, \alpha)$ is independent of ξ—that is, $t = t(\alpha)$.

Since q_0 is irreducible we have

$$\deg_\xi q(\xi) = k_0 \geq p.$$

Let us now consider the remainders of the monomials ξ^k (for $k \geq p$) after division by q_0—

$$\xi^k \equiv \lambda_0^{(k)}(\alpha) + \lambda_1^{(k)}(\alpha)\xi + \cdots + \lambda_{p-1}^{(k)}(\alpha)\xi^{p-1} \qquad (\text{mod } q_0).$$

We want to determine the degrees of the coefficients regarded as polynomials in α.

With the aid of the recursions

$$\begin{aligned} \lambda_0^{(k+1)}(\alpha) &= \frac{1 + \alpha^p}{2} \lambda_{p-1}^{(k)}(\alpha) \\ \lambda_i^{(k+1)}(\alpha) &= \lambda_{i-1}^{(k)}(\alpha) + \frac{1}{2}(-1)^i \binom{p}{i} \alpha^{p-i} \lambda_{p-1}^{(k)}(\alpha) \qquad (1 \leq i \leq p-1), \end{aligned}$$

an easy inductive argument shows that

$$\deg_\alpha \lambda_i^{(k)}(\alpha) \leq k - i.$$

Now set $\mu_i^{(k)} = 2^{k-p+1}\beta_i^{(k)}$, where $\beta_i^{(k)}$ denotes the coefficient of α^{k-i} in the polynomial $\lambda_i^{(k)}(\alpha)$. Then we have the recursions

$$\begin{aligned} \mu_0^{(k+1)} &= \mu_{p-1}^{(k)} \\ \mu_i^{(k+1)} &= 2\mu_{i-1}^{(k)} + (-1)^i \binom{p}{i} \mu_{p-1}^{(k)} \qquad (1 \leq i \leq p-1), \end{aligned}$$

from which it follows that the $\mu_i^{(k)}$ are integers. Also,

$$\mu_{p-1}^{(k+1)} = 2\mu_{p-2}^{(k)} + (-1)^{p-1}p\mu_{p-1}^{(k)} \equiv \mu_{p-1}^{(k)} \equiv \cdots \equiv \mu_{p-1}^{(p-1)} = 1 \pmod 2$$

and this implies that $\deg_\alpha \lambda_{p-1}^{(k)} = k - p + 1$. Since $t = t(\alpha)$ it follows, in particular, that

$$0 = \deg_\alpha \lambda_{p-1}^{(k_0)}(\alpha) = k_0 - p + 1 = \deg_\xi q(\xi) - p + 1,$$

and since $k_0 \geq p$, that is a contradiction. Hence q is constant, and the contradiction yields the desired conclusion for the case of odd p.

For even p, the basic idea of the proof is unchanged, but this time the leading coefficient of ξ^p in q_0 depends on α. Furthermore, the argument concerning the degree of the coefficient polynomials is different. Let α be prime and let α not divide p.

Let us again divide the monomials ξ^k (for $k \geq p-1$) by q_0, obtaining

$$\xi^{(k)} \equiv \lambda_0^{(k)}(\alpha) + \lambda_1^{(k)}(\alpha)\xi + \cdots + \lambda_{p-2}^{(k)}(\alpha)\xi^{p-2} \pmod{q_0}.$$

Observe that the degree of the remainder as a polynomial in ξ is now $p-2$ and that the coefficient functionals also contain powers of α^{-1}. We have the recursions

$$\lambda_0^{(k+1)}(\alpha) = \frac{1+\alpha^p}{p\alpha}\lambda_{p-2}^{(k)}(\alpha),$$

$$\lambda_i^{(k+1)}(\alpha) = \lambda_{i-1}^{(k)}(\alpha) + \frac{1}{p\alpha}(-1)^i\binom{p}{i}\alpha^{p-i}\lambda_{p-2}^{(k)}(\alpha) \qquad (1 \leq i \leq p-2),$$

and setting

$$\tilde\lambda_i^{(k)}(\alpha) = (p\alpha)^{k-p+2}\lambda_i^{(k)}(\alpha),$$

we obtain

$$\tilde\lambda_0^{(k+1)}(\alpha) = (1+\alpha^p)\tilde\lambda_{p-2}^{(k)}(\alpha),$$

$$\tilde\lambda_i^{(k+1)}(\alpha) = p\alpha\tilde\lambda_{i-1}^{(k)}(\alpha) + (-1)^i\binom{p}{i}\alpha^{p-i}\tilde\lambda_{p-2}^{(k)}(\alpha) \qquad (1 \leq i \leq p-2).$$

An easy inductive argument now shows that

$$\deg_\alpha \tilde\lambda_i^{(k)}(\alpha) \leq 2k - i - p + 2.$$

Now let $\tilde\mu_i^{(k)}$ denote the coefficient of $\alpha^{2k-i-p+2}$ in the polynomial $\tilde\lambda_i^{(k)}(\alpha)$. Then we have the recursions

$$\tilde\mu_0^{(k+1)} = \tilde\mu_{p-2}^{(k)}$$

$$\tilde\mu_i^{(k+1)} = p\tilde\mu_{i-1}^{(k)} + (-1)^i\binom{p}{i}\tilde\mu_{p-2}^{(k)} \qquad (1 \leq i \leq p-2),$$

which imply that the $\tilde\mu_i^{(k)}$ are integers.

Now consider the polynomial $s(\xi) = (1-\xi)^p - \xi^p$. The roots of s are $(1 + e^{(2\pi j/p)\Im})^{-1}$ for $j = 0, \cdots, p-1$. Let σ be any one of these roots. After some calculation we obtain

$$\sum_{i=0}^{p-2} \tilde\mu_i^{(k+1)}\sigma^i = p\sigma\sum_{i=0}^{p-2}\tilde\mu_i^{(k)}\sigma^i + ((1-\sigma)^p - \sigma^p)\,\tilde\mu_{p-2}^{(k)} = p\sigma\sum_{i=0}^{p-2}\tilde\mu_i^{(k)}\sigma^i,$$

and an inductive argument shows that

$$\sum_{i=0}^{p-2}\tilde\mu_i^{(k)}\sigma^i = p^{k-p+2}\sigma^k.$$

Setting $\sigma_1 = \frac{1}{2}, \sigma_2 = (1 + e^{2\pi\Im/p})^{-1}$, we obtain

$$\sum_{i=1}^{p-2} \tilde{\mu}_i^{(k)} (\sigma_1^i - \sigma_2^i) = p^{k-p+2}(\sigma_1^k - \sigma_2^k) \neq 0.$$

Thus for each k at least one of the $\tilde{\mu}_i$ is different from 0—say, $\tilde{\mu}_{i_0}^{(k)} \neq 0$. Then

$$\deg_\alpha \tilde{\lambda}_{i_0}^{(k)}(\alpha) = 2k - i_0 - p + 2.$$

Since $t = t(\alpha)$, it follows in particular that for $k_0 \geq p - 1$,

$$0 = \deg_\alpha \lambda_{i_0}^{(k_0)}(\alpha) = (2k_0 - i_0 - p + 2) - (k_0 - p + 2) = k_0 - i_0 = \deg_\xi q(\xi) - i_0,$$

which is a contradiction. Thus q is constant, completing the proof of Theorem 2.3. □

Now consider an arbitrary dimension $n \geq 3$ and let $u_1, \cdots, u_n$ denote the standard basis for $\mathbb{R}^n$. For each $\alpha \in \mathbb{N}$, let S_α denote the n-simplex whose vertices are $-u_1$, u_1, αu_2, $u_3, \cdots, u_n$. If $\mathbb{R}^n$ is equipped with a p-norm, then for all α it is true that $R_n(S_\alpha) = R(T_\alpha)$, where T_α is as in Lemma 2.1. It follows from Theorem 2.3 that when p is an integer ≥ 3, there is no rationalizing polynomial for R_n.

We next establish the algebraic intractability of the inradii of certain triangles. We begin with some calculations for a general $p \in \mathbb{N}$ (with $p \geq 2$), for they may be useful in extending the result on the badness of inradii to values of p other than 2. However, final considerations are restricted to the case $p = 2$. The triangles $W_\alpha =$ conv $\{(0,0), (1,0), (0,\alpha)\}$ in the following discussion are different from the triangles T_α used in discussing the circumradius, and in fact W_α would not work for the circumradius because for the p-norm the circumcenter of W_α is $(1/2, \alpha/2)$ and $R^p(W_\alpha)$ is the rational number $(1 + \alpha^p)/2$. Also, the triangles T_α do not work for the following proof of ρ's badness when $p = 2$, but perhaps they could be made to work by means of a different argument. (It turns out that

$$\rho(T_\alpha) = \frac{\alpha}{1 + (1 + \alpha^{p/(p-1)})^{(p-1)/p}}$$

so that

$$\rho(T_\alpha) = \frac{\alpha}{1 + \sqrt{1 + \alpha^2}}$$

when $p = 2$.)

LEMMA 2.4 (AN EQUATION SATISFIED BY CERTAIN INRADII). *For a fixed integer $p \in]1, \infty[$, and for each positive integer α, let ρ_α denote the inradius of the triangle W_α in $\mathbb{R}_p^2$ whose vertices are $(0,0), (1,0)$, and $(0,\alpha)$. Then*

$$\rho_\alpha = \alpha \left(1 + \alpha + \left(1 + \alpha^{p/(p-1)}\right)^{(p-1)/p}\right)^{-1},$$

and when $p = 2$,

$$\rho_\alpha = \frac{1}{2}\left(1 + \alpha - \sqrt{1 + \alpha^2}\right).$$

Proof. The incircle of W_α is unique and its center is $(\rho_\alpha, \rho_\alpha)$. The edge conv $\{(1,0), (0,\alpha)\}$ of W_α lies on the line given by $\alpha\xi + \eta = \alpha$. The distance of a point (ρ, ρ) from this line is the minimum (over $\xi \in \mathbb{R}$) of the function

$$\|(\rho, \rho) - (\xi, \alpha(1 - \xi))\|_p.$$

Since the pth power of $\| \ \|_p$ is a strictly convex functional, the only stationary point of the function $\|(\rho,\rho)-(\xi,\alpha(1-\xi))\|_p^p$ is its global minimum, and we obtain the relevant value of ξ by solving the equation,

$$p(\xi-\rho)^{p-1}-\alpha p(\alpha(1-\xi)-\rho)^{p-1} = 0.$$

Thus, we can compute ρ_α from the fact that

$$\left(\frac{\alpha^{1/(p-1)}(\alpha-\rho_\alpha)+\rho_\alpha}{1+\alpha^{p/(p-1)}}-\rho_\alpha\right)^p+\left(\alpha\left(1-\frac{\alpha^{1/(p-1)}(\alpha-\rho_\alpha)+\rho_\alpha}{1+\alpha^{p/(p-1)}}\right)-\rho_\alpha\right)^p = \rho_\alpha^p.$$

After some simplification we obtain

$$(\alpha-\rho_\alpha(1+\alpha))^p = \left(1+\alpha^{p/(p-1)}\right)^{p-1}\rho_\alpha^p,$$

which yields the stated conclusion. □

THEOREM 2.5 (COMPUTATIONAL INTRACTABLITY OF INRADII). *If $p=2$ and q is a nonconstant rational polynomial, then $q(\rho_\alpha)$ is rational for at most finitely many $\alpha\in\mathbb{N}$.*

Proof. For each $p\in]1,\infty[$ it is true that

$$\rho_\alpha \quad\longrightarrow\quad \frac{1}{2} \qquad \text{as} \qquad \alpha\to\infty.$$

From this it follows that if u is any polynomial for which $u(\rho_\alpha)$ is infinitely often an integer, then $u-u(\frac{1}{2})$ has infinitely many roots and thus u is constant. To complete the proof, it suffices to show that if $p=2$ and q is a rational polynomial such that $q(\rho_\alpha)$ is infinitely often rational, then there is a nonzero multiple $u=\mu q$ of q such that $u(\rho_\alpha)$ is infinitely often an integer.

Let the rational numbers κ_i be such that $q(\xi)=\sum_{i=0}^k\kappa_i\xi^i$. Then

$$q(\rho_\alpha)=\sum_{i=0}^k\kappa_i\frac{1}{2^i}(1+\alpha-\sqrt{1+\alpha^2})^i=\sum_{i=0}^k\frac{\kappa_i}{2^i}\sum_{j=0}^i\binom{i}{j}(1+\alpha)^j\sqrt{1+\alpha^2}^{\,i-j}.$$

Let the integer μ be such that $\mu\kappa_i/2^i$ is an integer for all i, and set $u=\mu q$. Then there are two polynomials $s(\alpha)$ and $t(\alpha)$ with integer coefficients such that

$$u(\rho_\alpha)=s(\alpha)+t(\alpha)\sqrt{1+\alpha^2}.$$

Since $u(\rho_\alpha)$ is infinitely often rational, $t(\alpha)\equiv 0$ and hence $u(\rho_\alpha)$ is infinitely often an integer. That completes the proof. □

Now consider an arbitrary dimension $n\geq 3$ and let $u_1,\cdots,u_n$ denote the standard basis for $\mathbb{R}^n$. Let V denote the set of all 2^{n-2} points of the form $\sum_{i=3}^n\epsilon_iu_i$ with $\epsilon_3,\cdots,\epsilon_n\in\{-1,1\}$. For each $\alpha\in\mathbb{N}$, let P_α denote the n-dimensional prism whose vertex-set is

$$V\cup(u_1+V)\cup(\alpha u_2+V).$$

If $\mathbb{R}^n$ is equipped with a p-norm then for each α it is true that $r_n(P_\alpha)=\rho(W_\alpha)$, where W_α is as in Lemma 2.4. It follows from Theorem 2.5 that when $p=2$ there is no rationalizing polynomial for r_n.

3. Two questions. We have seen in the preceding sections that the circumradius R is algebraically tractable in $\mathbb{M}_p$ for $p\in\{1,2,\infty\}$ but intractable for other $p\in\mathbb{N}$, while the inradius ρ is tractable in $\mathbb{M}_1$ and $\mathbb{M}_\infty$ but intractable in $\mathbb{M}_2$. An answer to the following question is needed to complete the picture. (We conjecture that the answer is negative.)

PROBLEM 3.1 (ALGEBRAIC TRACTABILITY OF INRADII). *Let p be a fixed integer greater than 2. Does there exist a nonconstant rational polynomial q such that $q(\rho(C))$ is rational whenever C is a convex lattice polygon in $\mathbb{M}_p$?*

The following problem rephrases, in a purely algebraic fashion, some of the problems that are suggested by the arguments of §2.

PROBLEM 3.2 (EXISTENCE OF RATIONALIZING POLYNOMIALS). *For each $s \in \mathbb{Z}[\xi, \eta]$, define*

$$N(s) = \{\xi \in [0, \infty[: \text{there exists an } \eta \in \mathbb{N} \text{ such that } s(\xi, \eta) = 0\}.$$

For which s does there exist a rationalizing polynomial, i.e., a polynomial $q \in \mathbb{Z}[\omega] \setminus \mathbb{Z}$ such that $q(\omega) \in \mathbb{Q}$ for all $\omega \in N(s)$?

In terms of this notation, some of the conclusions of §2 may be restated as follows: *Let $s_p(\xi, \eta) = \xi^p - (\eta - \xi)^p - 1$. Then s_p admits a rationalizing polynomial for $p = 1$ and $p = 2$ but not for any other $p \in \mathbb{N}$.*

REFERENCES

[1] C. BAJAJ, *The algebraic degree of geometric optimization problems*, Discrete Comput. Geom., 3 (1988), pp. 177–191.

[2] T. BONNESEN AND W. FENCHEL, *Theorie der konvexen Körper*, Springer-Verlag, Berlin, New York, 1934.

[3] H. EDELSBRUNNER, *Algorithms in Computational Geometry*, Springer-Verlag, Berlin, New York, 1987.

[4] P. GRITZMANN AND V. KLEE, *Inner and outer j-radii of convex bodies in finite-dimensional normed spaces*, Discrete Comput. Geom., to appear.

[5] ———, *Computational complexity of inner and outer j-radii of polytopes in finite-dimensional normed spaces*, to appear.

[6] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, Berlin, New York, 1985.

SIAM J. COMPUT.
Vol. 20, No. 2, p. 404, April 1991
© 1991 Society for Industrial and Applied Mathematics
014

ERRATUM: The Polynomial Time Hierarchy Collapses If The Boolean Hierarchy Collapses*

JIM KADIN†

Mahaney has pointed out that the proof of Lemma 4.8 on page 1275 of the original article [1] is incorrect and that the lemma is probably not true [3]. The lemma was used to establish how far the PH collapses under the assumption that the BH collapses.

Without Lemma 4.8, we had established that the collapse of the BH implies the existence of a sparse set S such that co-NP $\subseteq$ NP^S which, by the results of Yap [6], implies that PH $\subseteq \Sigma_3^P$.

Lemma 4.8 was then used to argue further that the sparse set S is itself in Σ_2^P, which would imply that the PH collapses still lower, down to $P^{NP^{NP}[O(\log n)]}$. It is not at all clear that the set S is in Σ_2^P, and hence the argument presented that the collapse of the BH implies PH $\subseteq P^{NP^{NP}[O(\log n)]}$ is invalid. Lemma 4.9 and Theorem 4.10 are the intermediate steps in this argument that depended on Lemma 4.8, hence their proofs are also invalid.

However, the overall result that the collapse of the BH implies PH $\subseteq P^{NP^{NP}[O(\log n)]}$ does follow from the results of Wagner [4]. Therefore all of the results stated in the paper, with the exception of Lemmas 4.8 and 4.9 and Theorem 4.10, are correct.

Wagner has since improved his results by showing that if the BH collapses to level k, then the PH is contained in $P^{NP^{NP}[O(1)]}$ [5]. Chang and Kadin more recently obtain the stronger conclusion that the collapse of the BH to level k implies PH $\subseteq$ $P^{NP^{NP}[k]}$ [2].

REFERENCES

[1] J. KADIN, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.

[2] R. CHANG AND J. KADIN, *The Boolean hierarchy and the polynomial hierarchy: A closer connection*, in Proc. 5th Structure in Complexity Theory Conference, July 1990, pp. 169–178.

[3] S. MAHANEY, Private communication, 1989.

[4] K. WAGNER, *Number-of-query hierarchies*, Tech. Report 158, University of Augsburg, Augsburg, Federal Republic of Germany, October 1987.

[5] ———, *Number-of-query hierarchies*, Tech. Report 4, Institut für Informatik, Universität Würzburg, Würzburg, Federal Republic of Germany, February 1989.

[6] C. YAP, *Some consequences of non-uniform conditions on uniform classes*, Theoret. Comput. Sci., 26 (1983), pp. 287–300.

* Received by the editors October 18, 1990; accepted for publication (in revised form) November 14, 1990.

† Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853. Present address, Department of Computer Science, Neville Hall, University of Maine, Orono, Maine 04469.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 405-422, June 1991
© 1991 Society for Industrial and Applied Mathematics
001

FAST MATCHING ALGORITHMS FOR POINTS ON A POLYGON*

ODILE MARCOTTE† AND SUBHASH SURI‡

Abstract. Given a set P of $2n$ points on the boundary of a polygon, consider the complete graph whose vertex set is P, and whose edges are assigned weights equal to the Euclidean distance between their endpoints if the endpoints see each other in the polygon and $+\infty$ otherwise. The problem of finding a minimum-weight perfect matching is investigated in this graph, and an $O(n \log(n))$ time algorithm is obtained if the polygon is convex; an $O(n \log^2(n))$ time algorithm is obtained if the polygon is simple but not convex. Similar results are obtained for the assignment problem and the maximum-weight problem.

Key words. geometric matching, nearest neighbors, divide and conquer, matrix searching, shortest paths, computational geometry

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U05

1. Introduction. Given an undirected graph $G=(V, E)$, a *matching* is a collection of vertex-disjoint edges of G. A matching is said to be *perfect* if it is a spanning subgraph of G. Not all graphs admit a perfect matching, however, and the problem of finding a matching of maximum cardinality is a classical problem of combinatorial optimization. Edmonds gave a polynomial-time algorithm for solving this problem in [4]. In a more general problem, we assign real-valued weights to the edges of G, and define the *weight* of a matching M to be the cumulative weight of all the edges of M. The problem of finding a *minimum-weight maximum-cardinality matching* was also solved by Edmonds [4], and his algorithm can be implemented in $O(|V|^3)$ time (see Lawler [7]).

The edge-weights of the graph induced by a set of points in the plane have a special structure. Indeed, several well-known problems can be solved more efficiently for graphs induced by such sets than for arbitrary graphs. For instance, a minimum spanning tree of n points of the plane can be computed in time $O(n \log(n))$, while the best algorithm known for arbitrary graphs takes $O(|E| \log^*(|V|))$ time. On the other hand, there are equally well-known problems for which the geometric nature of the problem does not help; for instance, both the combinatorial and geometric versions of the traveling salesman problem are NP-complete. It is natural, therefore, to ask whether the $O(|V|^3)$ time bound of Edmonds's algorithm can be substantially improved for the geometric version of the matching problem. (In this case, $|V|$ is the cardinality of the set of points under consideration.)

Let P be a collection of $2n$ points of the plane. Since the complete graph induced by P always contains a perfect matching, we will use the term "matching" instead of "perfect matching" in the following discussion. Thus a matching of P is a partition of its points into n pairs, each pair consisting of two distinct points. The weight of a pair is the Euclidean distance between its elements, and the weight of a matching is the sum of the weights of all the pairs in the matching. We consider the problem of finding a matching of minimum weight. Applications of this *geometric matching* problem

* Received by the editors November 28, 1988; accepted for publication (in revised form) April 11, 1990.

† Département de mathématiques et informatique, Université du Québec à Montréal, C. P. 8888, succ. "A," Montréal, Canada H3C 3P8. The research of this author was supported by National Sciences and Engineering Research Council of Canada grant A9126. Part of this work was done while she was visiting Bellcore.

‡ Bellcore, 415 South Street, Morristown, New Jersey 07960-1910.

are found in applied areas such as operations research, pattern recognition, statistics, and VLSI design (see, for example, Christofides [3], Koppe [6], and Werman et al. [10]).

Despite its apparent simplicity, very little of a qualitative nature is known about the geometric matching problem. Various conjectures relating minimum-weight matchings to minimum-weight triangulations, and other geometric structures, were made by Shamos [8], but counterexamples to all of them were found by Akl [2]. The best algorithm currently known for the geometric matching problem runs in $O(n^{5/2}\log^4(n))$ time (Vaidya [9]), while the best lower bound known is only $\Omega(n\log(n))$.

In this paper, we consider the geometric matching problem for sets of points that lie on the boundary of a convex polygon. We remark that Akl's counterexamples hold even in this restricted setting. We present an $O(n\log(n))$ time and $O(n)$ space algorithm for finding a minimum-weight matching in this case, where the number of points in the set is $2n$. It turns out that the key property required by our algorithm is "polygonization" and not convexity, and hence we can adapt our algorithm to the case of a simple nonconvex polygon. In the latter case, the weight of an edge is equal to the Euclidean distance between its endpoints if the endpoints see each other in the polygon, and $+\infty$ otherwise. The time complexity of our algorithm becomes $O(n\log^2(n))$ in this case.

Next we consider the bipartite version of the matching problem: given n red points and n blue points on the boundary of a convex polygon, find a minimum-weight matching in which each edge joins a red point to a blue point. Our algorithm for this problem (also called the *assignment problem*) also runs in time $O(n\log(n))$.

Last, we consider maximum-weight matchings. Finding a maximum-weight matching of points that lie on the boundary of a convex polygon is significantly easier than finding a minimum-weight matching. We give a linear-time algorithm for this problem. The same algorithm can be applied to the nonconvex case if we define the distance between two points to be the length of a shortest path between them in the polygon. Computing the weight of a maximum-weight matching in the nonconvex case takes $O(n\log(n))$ time.

The organization of the paper is as follows. The first four sections are devoted to the geometric matching problem for points on the boundary of a convex polygon. Sections 2 and 3 provide the theoretical foundation of our algorithm; in § 2 we prove the key geometric lemma of our paper, namely, the extensibility lemma, and in § 3 we introduce the notions of weighted-nearest-neighbor graph and critical edge. In § 4 we give an overview of the algorithm and explain how to update the weighted-nearest-neighbor graph as the algorithm removes vertices from the graph. At the end of § 4, we give a precise description of the algorithm, prove its correctness, and analyze its running time. In § 5 we show how our algorithm can be adapted to the case of a simple polygon that is not convex. In § 6 we consider the assignment problem. Section 7 is devoted to the maximum-weight matching problem. The paper concludes with § 8 where we discuss some open problems.

2. The extensibility lemma. Given a set of points $P=\{z_0, z_1, \cdots, z_{2n-1}\}$, where $z_0, z_1, \cdots, z_{2n-1}$ are ordered counterclockwise on the boundary of a convex polygon, we wish to compute a minimum-weight matching of P. For two points a and b of P, the edge (i.e., the line segment) between them is denoted ab. The *weight* of ab, denoted $d(a, b)$, is the Euclidean distance between a and b. The weight of a matching M, denoted $\phi(M)$, is the sum of the weights of all the edges in M, that is, $\phi(M)=\sum_{ab\in M} d(a, b)$. We often will use the term optimal matching instead of minimum-weight matching, and the term vertex instead of point.

The triangle inequality implies that the edges of an optimal matching do not intersect—for if edges ab and cd did intersect, we could replace them by ac and bd and thereby reduce the weight of the matching. We let $\{z_i, z_{i+1}, \cdots, z_j\}$ denote the portion of the boundary (in the counterclockwise sense) between z_i and z_j. The main result of this section is the following fact. If there exists an optimal matching of $\{z_l, z_{l+1}, \cdots, z_m\}$, say, N such that $z_l z_m \in N$, then there exists an optimal matching of P containing all the edges of N except perhaps $z_l z_m$. To formalize this statement, we introduce the following definition. A set N of vertex-disjoint edges is called *extensible* for P if there exists an optimal matching M of P such that $M \supseteq N$.

LEMMA 1 (extensibility lemma). *Let $\{z_l, z_{l+1}, \cdots, z_m\}$ be a contiguous subset of P containing an even number of points. If N is an optimal matching of $\{z_l, z_{l+1}, \cdots, z_m\}$ containing $z_l z_m$, then $N \setminus \{z_l z_m\}$ is extensible.*

Proof. We first prove the following claim, from which the lemma readily follows.

CLAIM. *Let M be a matching of P whose edges do not intersect. If M contains an edge $z_i z_j$ that intersects $z_l z_m$, then M is not an optimal matching of P.*

Proof of the Claim. Let $\{w_1 w'_1, w_2 w'_2, \cdots, w_k w'_k\}$ be the set of edges of M having one endpoint in $\{z_l, z_{l+1}, \cdots, z_m\}$ and the other in $\{z_{m+1}, \cdots, z_{l-1}\}$. By assumption, this set is not empty. Without loss of generality, we assume that the w_i belong to $\{z_l, z_{l+1}, \cdots, z_m\}$, the w'_i belong to $\{z_{m+1}, \cdots, z_{l-1}\}$, and $w_1, w_2, \cdots, w_k$ are ordered counterclockwise on the boundary of the polygon. Since the edges of M do not intersect, the points $w'_1, w'_2, \cdots, w'_k$ are ordered clockwise on the boundary. Finally, since $\{z_l, z_{l+1}, \cdots, z_m\}$ contains an even number of points, a simple parity argument shows that k is even.

Let a *pseudomatching* of P be a collection of edges such that all points of P except z_l and z_m are incident to exactly one edge, while z_l and z_m are incident to exactly two edges. We focus our attention on the pseudomatching $M_1 = M \cup \{z_l z_m\}$. Clearly,

$$\phi(M_1) = \phi(M) + d(z_l, z_m). \tag{1}$$

An *alternating cycle* with respect to a matching is an even cycle whose even-numbered edges belong to the matching and odd-numbered edges do not belong to it. We will show that there exists an alternating cycle C with respect to M_1 such that the symmetric difference of M_1 and C has smaller weight than M_1.

The choice of C depends on the values of the predicates "$z_l \in W$" and "$z_m \in W$," where $W = \{w_1, w_2, \cdots, w_k\}$. There are four possibilities: (1) $z_l \in W$ and $z_m \in W$; (2) $z_l \in W$ and $z_m \notin W$; (3) $z_l \notin W$ and $z_m \in W$; and (4) $z_l \notin W$ and $z_m \notin W$. The alternating cycle C for each of these cases is given below as an ordered list of vertices. (Case 2 is shown in Fig. 1; the other three cases are similar.)

(1) $C \equiv (z_l = w_1, w_2, w'_2, w'_3, \cdots, w'_{k-1}, w_{k-1}, w_k = z_m, z_l)$,

(2) $C \equiv (z_l = w_1, w_2, w'_2, w'_3, w_3, \cdots, w_k, w'_k, z_m, z_l)$,

(3) $C \equiv (z_l, w'_1, w_1, w_2, w'_2, \cdots, w'_{k-1}, w_{k-1}, w_k = z_m, z_l)$,

(4) $C \equiv (z_l, w'_1, w_1, w_2, w'_2, w'_3, \cdots, w_k, w'_k, z_m, z_l)$.

Since C is an alternating cycle with respect to M_1, the symmetric difference of M_1 and C (denoted M_2) is also a pseudomatching of P. We can express M_2 as the union of $M_1 \setminus C_1$ and C_2, where $C_1 \equiv C \cap M_1$ and $C_2 \equiv C \setminus C_1$. By using the triangle inequality and the fact that at least one edge of M intersects $z_l z_m$, we easily verify that $\phi(C_2) < \phi(C_1)$ and hence

$$\phi(M_2) < \phi(M_1). \tag{2}$$

By construction, every edge of M_2 is contained in either $\{z_l, z_{l+1}, \cdots, z_m\}$ or $\{z_m, z_{m+1}, \cdots, z_l\}$. Furthermore, of the two edges incident upon z_l (respectively, z_m),

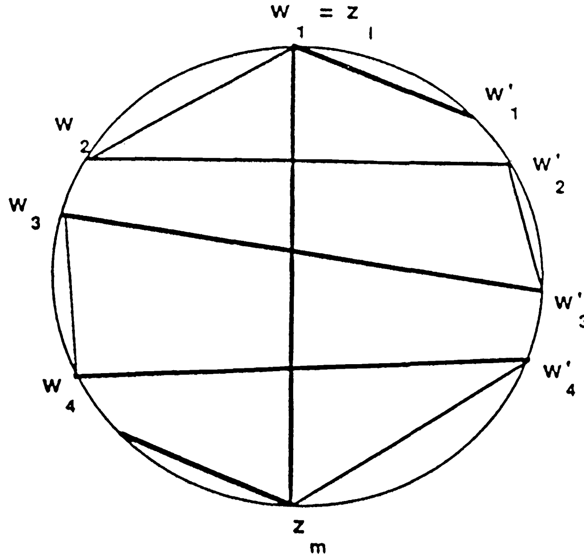


FIG. 1. *Alternating cycle of the extensibility lemma* (*Case* 2). *Matching edges are shown in heavy lines.*

one has an endpoint in $\{z_{l+1}, \cdots, z_{m-1}\}$ and the other an endpoint in $\{z_{m+1}, \cdots, z_{l-1}\}$. It follows that M_2 is the union of a matching of $\{z_l, z_{l+1}, \cdots, z_m\}$ and a matching of $\{z_m, z_{m+1}, \cdots, z_l\}$. We obtain the following relations:

$$\begin{aligned}
\phi(M_2) &= \phi(N_1)+\phi(N_2) && \text{where } N_1 \text{ is a matching of } \{z_l, z_{l+1}, \cdots, z_m\} \text{ and } N_2 \\
& && \text{a matching of } \{z_m, z_{m+1}, \cdots, z_l\} \\
&\geqq \phi(N)+\phi(N_2) && \text{since } N \text{ is an optimal matching of } \{z_l, z_{l+1}, \cdots, z_m\} \\
&= \phi(N\backslash\{z_l z_m\})+\phi(N_2)+d(z_l, z_m).
\end{aligned} \tag{3}$$

By combining (1)-(3), we conclude that the weight of M is greater than the weight of $(N\backslash\{z_l z_m\}) \cup N_2$. Since $(N\backslash\{z_l z_m\}) \cup N_2$ is a matching of P, M cannot be an optimal matching of P. This completes the proof of the claim.

Let us now consider an optimal matching M of P. The edges of M do not intersect, and the preceding claim thus implies that no edge of M intersects $z_l z_m$. Hence for any edge $z_i z_j \in M$, either z_i and z_j both belong to $\{z_l, z_{l+1}, \cdots, z_m\}$ or they both belong to $\{z_m, z_{m+1}, \cdots, z_l\}$. Therefore M is either the union of a matching of $\{z_l, z_{l+1}, \cdots, z_m\}$ and a matching of $\{z_{m+1}, \cdots, z_{l-1}\}$, or the union of a matching of $\{z_{l+1}, \cdots, z_{m-1}\}$ and a matching of $\{z_m, z_{m+1}, \cdots, z_l\}$. In the former case we may assume, without loss of generality, that $M \supseteq N$, since N is an optimal matching of $\{z_l, z_{l+1}, \cdots, z_m\}$. In the latter case, we may assume that $M \supseteq N\backslash\{z_l z_m\}$, since $N\backslash\{z_l z_m\}$ is an optimal matching of $\{z_{l+1}, \cdots, z_{m-1}\}$. In both cases, we conclude that the set $N\backslash\{z_l z_m\}$ is extensible. This completes the proof of the lemma. □

We use the extensibility lemma to replace a set P by a smaller set P'. The following remark states that the procedure applied to P can also be applied to P'.

Remark 1. Let N be an extensible set of edges for P, and P' the set of points that remain after deleting the endpoints of the edges of N. If N' is an extensible set for P', then $N \cup N'$ is extensible for P.

3. Vertex weights and boundary matchings. We now introduce the notion of "vertex weights," which are real numbers assigned to the points of P. Let us label the points of P as $x_0, y_0, x_1, \cdots, x_{n-1}, y_{n-1}$ in a counterclockwise traversal of the boundary of the polygon. Since the edges of an optimal matching are pairwise disjoint, only the edges of the form x_iy_j are candidates for inclusion in an optimal matching. We assign weights u_i to x_i and v_j to y_j as follows:

$$\begin{aligned} &u_0 = 0, \\ &v_i = d(x_i, y_i) - u_i \quad \text{for } i = 0, 1, \cdots, n-1, \quad \text{and} \\ &u_i = d(y_{i-1}, x_i) - v_{i-1} \quad \text{for } i = 1, \cdots, n-1. \end{aligned} \tag{4}$$

Thus the weights of any two adjacent points on the boundary of the polygon sum up to their distance, with the possible exception of the pair $\{x_0, y_{n-1}\}$. In Lemma 2 below we prove a useful equality relating this pair to the weights of the two "boundary" matchings of P. Let M_1 and M_2 denote these matchings: $M_1 = \{x_0y_0, x_1y_1, \cdots, x_{n-1}y_{n-1}\}$ and $M_2 = \{y_0x_1, y_1x_2, \cdots, y_{n-1}x_0\}$.

LEMMA 2. *If the points of P are assigned weights as in* (4), *then $\phi(M_1) - \phi(M_2)$ is equal to $u_0 + v_{n-1} - d(y_{n-1}, x_0)$.*

Proof. The lemma is a simple consequence of the following equalities:

$$\begin{aligned} u_0 + v_{n-1} &= \sum_{i=0}^{n-1} (u_i + v_i) - \sum_{i=1}^{n-1} (v_{i-1} + u_i) \\ &= \sum_{i=0}^{n-1} d(x_i, y_i) - \sum_{i=1}^{n-1} d(y_{i-1}, x_i) \\ &= \sum_{i=0}^{n-1} d(x_i, y_i) - \sum_{i=0}^{n-1} d(y_{i-1}, x_i) + d(y_{n-1}, x_0) \\ &= \phi(M_1) - \phi(M_2) + d(y_{n-1}, x_0). \end{aligned}$$

This completes the proof of the lemma. □

We use the weights defined above to introduce the notion of weighted distance. The *weighted distance* between x_i and y_j is defined as $D(x_i, y_j) = d(x_i, y_j) - u_i - v_j$. Then we have the following lemma.

LEMMA 3 (difference lemma). *Let x_k and y_l be two nonadjacent points on the boundary of the polygon. Let*

$$N_1 = \{x_ky_l, y_kx_{k+1}, \cdots, y_{l-1}x_l\}$$

and $N_2 = \{x_ky_k, x_{k+1}y_{k+1}, \cdots, x_ly_l\}$ if $k < l$, and let $N_1 = \{x_ky_l, x_{l+1}y_{l+1}, \cdots, x_{k-1}y_{k-1}\}$ and $N_2 = \{y_lx_{l+1}, y_{l+1}x_{l+2}, \cdots, y_{k-1}x_k\}$ otherwise. Then $\phi(N_1) - \phi(N_2) = D(x_k, y_l)$.

Proof. We only prove the lemma for $k < l$; the other case is similar.

$$\begin{aligned} \phi(N_1) - \phi(N_2) &= \sum_{i=k}^{l-1} d(y_i, x_{i+1}) + d(x_k, y_l) - \sum_{i=k}^{l} d(x_i, y_i) \\ &= \sum_{i=k}^{l-1} (v_i + u_{i+1}) + d(x_k, y_l) - \sum_{i=k}^{l} (u_i + v_i) \\ &= d(x_k, y_l) - u_k - v_l \\ &= D(x_k, y_l). \end{aligned}$$

This completes the proof. □

We now introduce the notion of weighted-nearest neighbor. A *weighted-nearest neighbor* of x_i is a point $y_{j(i)} \in \{y_0, y_1, \cdots, y_{n-1}\}$ such that $D(x_i, y_{j(i)}) = \min\{D(x_i, y_k) \mid k = 0, 1, \cdots, n-1\}$. A weighted-nearest neighbor of y_j among x_i's is defined in a similar fashion. The following lemma ties the notion of weighted-nearest neighbor to that of optimal matching.

LEMMA 4 (optimality lemma).

- $M_1 = \{x_0y_0, x_1y_1, \cdots, x_{n-1}y_{n-1}\}$ *is an optimal matching of* P *if* $D(x_i, y_{j(i)}) \geqq 0$ *for all* $i \in \{0, 1, \cdots, n-1\}$.
- $M_2 = \{y_0x_1, y_1x_2, \cdots, y_{n-1}x_0\}$ *is an optimal matching of* P *if* y_{n-1} *is a weighted-nearest neighbor of* x_0 *with* $D(x_0, y_{n-1}) \leqq 0$, *and* $D(x_i, y_{j(i)}) \geqq 0$ *for all* $i \in \{1, 2, \cdots, n-1\}$.

Proof. We only prove the first part of the lemma, since the second part is similar. Let us first assume that M_1 is not optimal. If M_2 is optimal, then $\phi(M_1) - \phi(M_2) > 0$, and by Lemma 2, $D(y_{n-1}, x_0) = d(y_{n-1}, x_0) - u_0 - v_{n-1} < 0$. If neither M_1 nor M_2 is optimal, we let M denote an optimal matching of P containing the maximum number of "boundary edges," i.e., edges joining two adjacent points of P. We observe that M contains a "diagonal edge," i.e., an edge whose endpoints are nonadjacent on the boundary of the polygon.

Among all diagonal edges, we choose one, say, x_ky_l that is "shallow," and assume, without loss of generality, that $k < l$. Then the restriction of M to $\{x_k, y_k, \cdots, x_l, y_l\}$ is $N_1 = \{x_ky_l, y_kx_{k+1}, \cdots, y_{l-1}x_l\}$. Since M is an optimal maching of P, it follows that N_1 is an optimal matching of $\{x_k, y_k, \cdots, x_l, y_l\}$. If we let N_2 denote the matching $\{x_ky_k, x_{k+1}y_{k+1}, \cdots, x_ly_l\}$, then by the difference lemma $D(x_k, y_l) = \phi(N_1) - \phi(N_2) \leqq 0$. If $D(x_k, y_l)$ were equal to zero, the matching $(M \setminus N_1) \cup N_2$ would be optimal for P but have fewer diagonals (and hence more boundary edges) than M. But that is impossible by our choice of M, and so we must have $D(x_k, y_l) < 0$. This completes the proof in the forward direction. □

Next we explore the case where neither M_1 nor M_2 is optimal. We begin by defining the notion of weighted-nearest-neighbor graph. The *weighted-nearest-neighbor graph* of P, denoted $G_w(P)$, is the (bipartite) graph whose nodes are the points of P and whose edges are of the form $x_iy_{j(i)}$ for $i = 0, 1, \cdots, n-1$. A simple argument based on the triangle inequality shows that the straight-line embedding of $G_w(P)$ is outerplanar.

We call an edge x_ky_l of $G_w(P)$ (where $x_ky_l \neq x_0y_{n-1}$) a *critical edge* if the following conditions hold:

- $D(x_k, y_l) < 0$, and
- if $k < l$, then $D(x_i, y_{j(i)}) \geqq 0$ for all $i \in \{k+1, \cdots, l\}$, and otherwise $D(x_i, y_{j(i)}) \geqq 0$ for all $i \in \{l+1, \cdots, k-1\}$.

(Note that if x_ky_l is a critical edge, then either $k < l$ or $k > l+1$. This follows from the choice of the weights: $D(x_i, y_j) = 0$ whenever x_i and y_j are adjacent and $x_iy_j \neq x_0y_{n-1}$.)

We observe that since $G_w(P)$ is outerplanar, either $D(x_i, y_{j(i)}) \geqq 0$ for all i, $i \neq 0$, or $G_w(P)$ contains at least one critical edge. Thus whenever the optimality lemma cannot be applied to conclude that M_1 or M_2 is optimal, the following lemma may be used to find an extensible set of edges.

LEMMA 5. *Let* x_ky_l *be a critical edge of* $G_w(P)$. *If* $k < l$, *then the set* $\{y_kx_{k+1}, \cdots, y_{l-1}x_l\}$ *is extensible for* P. *Otherwise, the set* $\{x_{l+1}y_{l+1}, \cdots, x_{k-1}y_{k-1}\}$ *is extensible for* P.

Proof. We give the proof for $k < l$; the other case is similar. Let N_1 denote $\{x_ky_l, y_kx_{k+1}, \cdots, y_{l-1}x_l\}$ and N_2 denote $\{x_ky_k, x_{k+1}y_{k+1}, \cdots, x_ly_l\}$. We prove that N_1 is an optimal matching of $\{x_k, y_k, \cdots, x_l, y_l\}$. The lemma then follows from the extensibility lemma.

Let N be an optimal matching of $\{x_k, y_k, \cdots, x_l, y_l\}$. Without loss of generality, we may assume that N contains the maximum number of boundary edges. If the number of boundary edges is less than $l-k+1$, then N contains at least one diagonal edge. Let $x_p y_q$ be a shallow diagonal, i.e., a diagonal such that each point of $\{y_p, x_{p+1}, \cdots, x_q\}$ (if $p<q$) or $\{x_{q+1}, y_{q+1}, \cdots, y_{p-1}\}$ (if $q<p-1$) is matched to an adjacent point on the boundary. There are two cases to consider: either $p \neq k$ or $p=k$. We claim that in both cases, N can be replaced by a matching containing fewer diagonals (and hence more boundary edges) than N.

Suppose first that $p \neq k$. If $p<q$, let $N_3=\{x_p y_q, y_p x_{p+1}, \cdots, y_{q-1} x_q\}$ and $N_4=\{x_p y_p, x_{p+1} y_{p+1}, \cdots, x_q y_q\}$. Otherwise (i.e., if $q<p-1$), let

$$N_3=\{x_p y_q, x_{q+1} y_{q+1}, \cdots, x_{p-1} y_{p-1}\}$$

and $N_4=\{y_q x_{q+1}, y_{q+1} x_{q+2}, \cdots, y_{p-1} x_p\}$. Since $x_k y_l$ is critical, $D(x_p, y_q) \geqq 0$ and the difference lemma implies that $\phi(N_3)-\phi(N_4) \geqq 0$. The weight of the matching $(N \setminus N_3) \cup N_4$ is therefore no greater than that of N, and it contains fewer diagonal edges than N. This contradicts the choice of N, and hence there cannot be a diagonal of the form $x_p y_q$ for $p \neq k$.

Let us now suppose that $p=k$. Clearly, N must be of the form $\{x_k y_q, y_k x_{k+1}, \cdots, y_{q-1} x_q, x_{q+1} y_{q+1}, \cdots, x_l y_l\}$. By a calculation similar to the one in the difference lemma, we show that $\phi(N_1)-\phi(N) \leqq 0$:

$$\begin{aligned}
\phi(N_1)-\phi(N) &= \sum_{i=k}^{l-1} d(y_i, x_{i+1}) + d(x_k, y_l) - \sum_{i=k}^{q-1} d(y_i, x_{i+1}) - d(x_k, y_q) - \sum_{i=q+1}^{l} d(x_i, y_i) \\
&= \sum_{i=q}^{l-1} d(y_i, x_{i+1}) + d(x_k, y_l) - d(x_k, y_q) - \sum_{i=q+1}^{l} d(x_i, y_i) \\
&= \sum_{i=q}^{l-1} (v_i + u_{i+1}) + D(x_k, y_l) + u_k + v_l - D(x_k, y_q) - u_k - v_q - \sum_{i=q+1}^{l} (u_i + v_i) \\
&= D(x_k, y_l) - D(x_k, y_q) \\
&\leqq 0 \quad \text{since } y_l \text{ is a weighted-nearest neighbor of } x_k.
\end{aligned}$$

This shows that if N contained any diagonal, it could be replaced by an optimal matching with fewer diagonals, contradicting the choice of N. Therefore N must be a boundary matching, and since the difference lemma implies that $\phi(N_1)<\phi(N_2)$, N must equal N_1. This completes the proof of the lemma. □

A naive application of Lemmas 4 and 5 is likely to be inefficient, since each invocation of Lemma 5 adds some edges to the final matching and deletes their endpoints from P. In general, deletion of points invalidates the vertex-weights and requires updating the nearest-neighbor graph. These updates may be expensive and since the number of edges found each time may be small, the time complexity of a naive algorithm will be quadratic or worse. In the following section, we show how the divide-and-conquer approach yields an $O(n \log(n))$ algorithm for the geometric matching problem.

4. An $O(n \log(n))$ algorithm for the geometric matching problem.

4.1. Divide and conquer. Our algorithm uses the divide-and-conquer paradigm. We divide the set P into two nearly equal halves: $P_1=\{x_0, y_0, \cdots, x_{\lfloor n/2 \rfloor}, y_{\lfloor n/2 \rfloor}\}$ and $P_2=\{x_{\lfloor n/2 \rfloor+1}, y_{\lfloor n/2 \rfloor+1}, \cdots, x_{n-1}, y_{n-1}\}$. We then recursively find optimal matchings of P_1 and P_2, say, N_1 and N_2, respectively. Let E_1 (respectively, E_2) be the set of edges removed by applications of Lemma 5 in the course of solving the problem for P_1

(respectively, P_2). Consider the outerplanar graph whose vertex set is $P_1 \cup P_2$ and whose edge set is $E_1 \cup E_2 \cup \{\text{boundary edges of } P\}$. Let N_{12} be the set of matching edges in this graph that do not lie on the same face as $x_0 y_{n-1}$. Then by the extensibility lemma, N_{12} is extensible for P. Therefore it remains to find an optimal matching for the set of points that belong to the same face as $x_0 y_{n-1}$. In other words, let R (respectively, L) denote the subset of P_1 (respectively, P_2) that remains after deleting the endpoints of edges of N_{12}. Then the conquer step of our algorithm consists of computing an optimal matching for $L \cup R$.

An instance of this situation is illustrated in Fig. 2, where the algorithm has been applied to sets $\{z_0, z_1, \cdots, z_7\}$ and $\{z_8, z_9, \cdots, z_{15}\}$. The edges of the two optimal matchings are shown in heavy lines, and $N_{12} = \{z_1 z_2, z_5 z_6, z_{11} z_{12}, z_{13} z_{14}\}$ is an extensible set of edges.

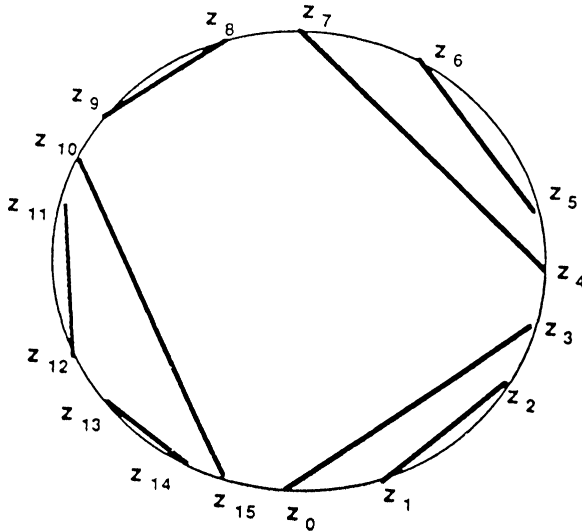


FIG. 2. *Illustration of the divide step in the algorithm.*

We relabel the points so that $R = \{x_0, y_0, \cdots, x_{m-1}, y_{m-1}\}$ and $L = \{x_m, y_m, \cdots, x_{r-1}, y_{r-1}\}$. Next we assign weights to the points of $L \cup R$:

$$u_0 = 0$$

$$v_i = d(x_i, y_i) - u_i \qquad \text{for } i = 0, 1, \cdots, r-1, \quad \text{and}$$

$$u_i = d(y_{i-1}, x_i) - v_{i-1} \quad \text{for } i = 1, \cdots, r-1.$$

Recalling the definition of weighted distance (see § 3), we say that the edge $x_i y_j$ is negative if $D(x_i, y_j) < 0$. We observe that, by construction, $\{x_0 y_0, x_1 y_1, \cdots, x_{m-1} y_{m-1}\}$ and $\{x_m y_m, x_{m+1} y_{m+1}, \cdots, x_{r-1} y_{r-1}\}$ are optimal matchings of R and L, respectively. Furthermore, $D(x_i, y_j) \geqq 0$ whenever x_i and y_j both belong to R or L. Thus any negative edge must have one endpoint in L and the other in R. This observation allows us to capture all the negative edges by computing weighted-nearest neighbors of points of R (respectively, L) among points of L (respectively, R).

During the conquer phase we use the following restricted definition of a weighted-nearest neighbor: for $x_i \in R$ (respectively, $x_i \in L$), the weighted-nearest neighbor of x_i is a point $y_{j(i)}$ of L (respectively, R) whose weighted distance from x_i is as small as possible. Let G_1 (respectively, G_2) be the graph whose vertex set is $L \cup R$ and whose

edges are of the form $x_iy_{j(i)}$ for $i=0, 1, \cdots, m-1$ (respectively, for $i=m, m+1, \cdots, r-1$). We use $G_1 \cup G_2$ to denote the union of G_1 and G_2. The following lemma formalizes the observation that $G_1 \cup G_2$ includes all the negative edges.

LEMMA 6. *Let $G_w(L\cup R)$ be the weighted-nearest-neighbor graph of $L\cup R$, as defined in* § 3. *The edges of $G_w(L\cup R)$ are of the form $x_iy_{j(i)}$, where $D(x_i, y_{j(i)}) = \min\{D(x_i, y_k) \mid k=0, 1, \cdots, r-1\}$. Let $G_1\cup G_2$ be defined as above. Then $G_w(L\cup R)$ and $G_1 \cup G_2$ have the same set of negative edges.*

Proof. We first prove that $D(x_i, y_j) \geqq 0$ whenever x_i and y_j both belong to R or L. As far as R is concerned, $D(x_i, y_j) \geqq 0$ for all x_i, $y_j \in R$ by construction of $\{x_0y_0, x_1y_1, \cdots, x_{m-1}y_{m-1}\}$. To prove the claim for L, let us assign weights to its points as follows:

$$u'_m = 0,$$

$$v'_i = d(x_i, y_i) - u'_i \qquad \text{for } i=m, m+1, \cdots, r-1, \quad \text{and}$$

$$u'_i = d(y_{i-1}, x_i) - v'_{i-1} \quad \text{for } i=m+1, \cdots, r-1.$$

Then we easily verify that $u'_i = u_i - u_m$ and $v'_i = v_i + u_m$ for $i = m, m+1, \cdots, r-1$. For a pair x_i, $y_j \in L$, the new weighted distance is $d(x_i, y_j) - u'_i - v'_j = d(x_i, y_j) - u_i - v_j = D(x_i, y_j)$. Again, we have $D(x_i, y_j) \geqq 0$.

Let us now consider a negative edge x_ky_l of $G_w(L\cup R)$. Since $D(x_k, y_l) < 0$, it follows from the above discussion that x_k and y_l must be in different sets. Thus x_ky_l is also an edge of $G_1\cup G_2$. Conversely, let x_ky_l be a negative edge of $G_1 \cup G_2$. We have either

(i) $x_k \in R$, $y_l \in L$, and $D(x_k, y_l) = \min_{m\leqq j\leqq r-1} D(x_k, y_j)$, or

(ii) $x_k \in L$, $y_l \in R$, and $D(x_k, y_l) = \min_{0\leqq j \leqq m-1} D(x_k, y_j)$.

In both cases, it easily follows that x_ky_l also belongs to $G_w(L\cup R)$. This completes the proof. □

The above lemma, in conjunction with the optimality lemma, says that $\{x_0y_0, x_1y_1, \cdots, x_{r-1}y_{r-1}\}$ is optimal for $L\cup R$ if no edge of $G_1\cup G_2$ is negative, and that $\{y_0x_1, y_1x_2, \cdots, y_{r-1}x_0\}$ is optimal for $L\cup R$ if x_0y_{r-1} is the only negative edge of $G_1\cup G_2$.

We observe that the negative edges of $G_1\cup G_2$ form an outerplanar graph (recall that $G_w(L\cup R)$ is outerplanar). This allows us to define a *total* order on these edges in the following way. Suppose we place the polygon such that the line separating L and R is vertical, L is on the left, R is on the right, and $y_{m-1}x_m$ is at the top. Then we say that x_ky_l precedes x_iy_j if x_ky_l is above x_iy_j. Let x_ky_l be the negative edge that is closest to $y_{m-1}x_m$, i.e., the *smallest* edge in the total order. Then it is clear that x_ky_l is a critical edge (cf. § 3) and, by Lemma 5, the set of edges $\{y_kx_{k+1}, \cdots, y_{l-1}x_l\}$ (if $x_k\in R$) or $\{x_{l+1}y_{l+1}, \cdots, x_{k-1}y_{k-1}\}$ (if $x_k \in L$) is extensible. Thus we can add these edges to the optimal matching, and remove their endpoints from further consideration. Deleting the endpoints reduces the size of the problem, but may invalidate vertex-weights and nearest-neighbor relations. The following discussion addresses these problems.

4.2. Updating weighted-nearest-neighbor graphs. Let x_ky_l be the smallest negative edge of $G_1\cup G_2$ in the total order of negative edges. We define the *breakthrough at* x_ky_l as the process of recognizing x_ky_l, adding the corresponding set of edges to the optimal matching and deleting their endpoints from $L\cup R$. Recall that the extensible matching associated with x_ky_l is $\{y_kx_{k+1}, \cdots, y_{l-1}x_l\}$ if $x_k \in R$, and $\{x_{l+1}y_{l+1}, \cdots, x_{k-1}y_{k-1}\}$ if $x_k \in L$. Let L' (respectively, R') be the set of points of L

(respectively, R) that remain after we delete the endpoints of these edges. That is, $L'=\{y_l, x_{l+1}, \cdots, y_{r-1}\}$ if $x_k \in R$, and $L'=\{x_k, y_k, \cdots, y_{r-1}\}$ if $x_k \in L$. We need to update the vertex-weights of points of $L' \cup R'$ in order to satisfy the constraints of (4); note in particular that although x_k and y_l are adjacent on the boundary of $L' \cup R'$, the sum of their weights may not be equal to $d(x_k, y_l)$.

In the following discussion, we will consistently use primed symbols to denote entities modified by a breakthrough, and unprimed symbols to denote the corresponding entities before the breakthrough. For instance, u_i' and v_j' denote the respective weights of x_i and y_j after a breakthrough, where x_i and y_j are points of $L' \cup R'$. The following lemma describes how to update the vertex-weights.

LEMMA 7 (update lemma). *Let $x_k y_l$ be the smallest negative edge in the total order defined above, and L' and R' the subsets of L and R that remain after the breakthrough at $x_k y_l$. The weights are unchanged for vertices of R', that is, $u_i' = u_i$ and $v_j' = v_j$ for all $x_i, y_j \in R'$. If δ denotes $u_k + v_l - d(x_k, y_l)$, then the new weights of vertices of L' are given by*

$$u_i' = u_i + \delta \quad \text{and} \quad v_j' = v_j - \delta \quad \text{if } x_k \in R, \quad \text{and}$$

$$u_i' = u_i - \delta \quad \text{and} \quad v_j' = v_j + \delta \quad \text{if } x_k \in L.$$

Proof. In the counterclockwise traversal of the boundary starting at x_0, the points of R' precede the first point removed during the breakthrough at $x_k y_l$. Hence the weights of the points of R' are not modified by the breakthrough, i.e., $u_i' = u_i$ for $x_i \in R'$ and $v_j' = v_j$ for $y_j \in R'$. We now consider the points of L', and assume that $x_k \in R$. (The other case is similar.) Since x_k also belongs to R', u_k' is equal to u_k. On the other hand, since y_l is adjacent to x_k on the boundary of $L' \cup R'$, the new weight of y_l is $v_l' = d(x_k, y_l) - u_k' = d(x_k, y_l) - u_k$. From this we deduce that $v_l - v_l' = u_k + v_l - d(x_k, y_l)$, that is, the weight of y_l decreases by δ during the breakthrough at $x_k y_l$. Since $u_i + v_j = d(x_i, y_j)$ for every edge of the boundary between y_l and y_{r-1}, we have $v_j' = v_j - \delta$ for $y_j \in L'$ and $u_i' = u_i + \delta$ for $x_i \in L'$. This completes the proof of the lemma. □

We now examine the effect of these weight modifications on nearest-neighbor relations. Let $D'(x_i, y_j) = d(x_i, y_j) - u_i' - v_j'$, and let G_1' and G_2' denote the weighted-nearest-neighbor graphs after the breakthrough at $x_k y_l$. (Note that both of these graphs are defined on the vertex set $L' \cup R'$, and that their edges are computed by using the new vertex-weights.) Given a point x_i of $L' \cup R'$, there are two cases to consider: either the weighted-nearest neighbor of x_i belongs to $L' \cup R'$, or it does not. Lemmas 8 and 9 deal with these two cases.

LEMMA 8 (retaining lemma). *Let $x_i y_j$ be an edge of $G_1 \cup G_2$ both of whose endpoints belong to $L' \cup R'$. Then $x_i y_j$ is also an edge of $G_1' \cup G_2'$.*

Proof. We shall only consider the case where x_k belongs to R, since the other case is similar. Let us assume first that $x_i \in R$. Then $x_i y_j$ is an edge of G_1 and $D(x_i, y_j) = \min\{D(x_i, y_q) \mid q = m, m+1, \cdots, r-1\}$. For any $y_q \in L'$, the update lemma implies that

$$D(x_i, y_q) + \delta = d(x_i, y_q) - u_i - v_q + \delta = d(x_i, y_q) - u_i' - v_q' = D'(x_i, y_q).$$

Thus the weighted distances from x_i to points of L' all change by the same amount, and y_j remains a nearest neighbor of x_i, implying that $x_i y_j \in G_1'$. A similar argument applies if $x_i \in L$. □

We now turn to the important case, namely, the one where x_i belongs to R' and $y_{j(i)}$ to $L \backslash L'$ or x_i belongs to L' and $y_{j(i)}$ to $R \backslash R'$. (Recall that $y_{j(i)}$ denotes the weighted-nearest neighbor of x_i.) In this case, we might think that a new weighted-nearest neighbor of x_i must be computed. If this computation were necessary, the

divide-and-conquer algorithm could take time proportional to n^2. Fortunately, we can show that x_i will not be the endpoint of a negative edge during any of the subsequent breakthroughs, which allows the algorithm to overlook the "dangling" vertices such as x_i.

LEMMA 9 (discarding lemma). *Let G_1 and G_2 (respectively, G_1' and G_2') denote the weighted-nearest-neighbor graphs before (respectively, after) the breakthrough at $x_k y_l$. Let $x_i y_j$ be an edge of $G_1 \cup G_2$ such that x_i belongs to R' and y_j to $L \backslash L'$, or x_i belongs to L' and y_j to $R \backslash R'$. Then after each of the subsequent breakthroughs the weighted distance between x_i and its weighted-nearest neighbor is nonnegative.*

Proof. We shall assume that $x_k y_l$ belongs to G_1. (The other case is similar.) We first observe that $x_i \notin R$, since that would imply that $x_i y_j$ and $x_k y_l$ intersect, contradicting the planarity of G_1. So $x_i \in L$ and the indices satisfy $l < i$ and $k \leqq j$. (See Fig. 3 for an illustration of this proof.) Consider an arbitrary series of breakthroughs that does not result in the deletion of x_i. Since the breakthrough edges induce a planar graph, we may label them $x_{k(1)}y_{l(1)}, x_{k(2)}y_{l(2)}, \cdots, x_{k(p)}y_{l(p)}$ (where $x_{k(1)}y_{l(1)} = x_k y_l$), in such a way that the labeling is consistent with the total order of negative edges. There are two types of breakthrough edges: those in which $x_{k(t)} \in R$, and those in which $x_{k(t)} \in L$. Let s denote the largest index such that $x_{k(s)} \in R$. (Observe that such an index always exists.)

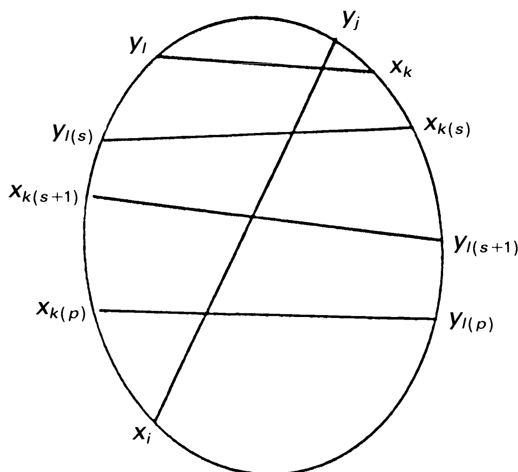


FIG. 3. *Illustration of the discarding lemma.*

We now prepare to set up inequalities that yield the desired result. Let u_i denote the weight of x_i before the breakthrough at $x_k y_l$, u_i' the weight of x_i after the breakthrough at $x_{k(s)}y_{l(s)}$, and u_i^* the weight of x_i after the breakthrough at $x_{k(p)}y_{l(p)}$. We use similar notation for the other points. If $x_{k(s)}y_{l(s)} = x_{k(p)}y_{l(p)}$, then obviously $u_i^* = u_i'$. If $s < p$, then edges $x_{k(t)}y_{l(t)}$ for $s < t \leqq p$ are such that $x_{k(t)} \in L$ (see Fig. 3). We observe that

$$u_i^* \leqq u_i'. \tag{5}$$

This is implied by the update lemma, since the breakthrough at $x_{k(t)}y_{l(t)}$ (for every t such that $s < t \leqq p$) will decrease the value of u_i'.

Next, since $x_{k(s)} \in R$ and $y_{l(s)} \in L$, we have $u'_{k(s)} = u_{k(s)}$ and $v'_{l(s)} = d(x_{k(s)}, y_{l(s)}) - u_{k(s)}$. Hence the weight of $y_{l(s)}$ has decreased by

$$\delta' = v_{l(s)} - v'_{l(s)} = u_{k(s)} + v_{l(s)} - d(x_{k(s)}, y_{l(s)}). \tag{6}$$

Accordingly, the weight of x_i has increased by δ' after the breakthrough at $x_{k(s)}y_{l(s)}$, and we obtain

(7) $$u'_i = u_i + \delta'.$$

Let y_q be any point of R that remains after completion of all the breakthroughs, including the one at $x_{k(p)}y_{l(p)}$. Since y_j is a weighted-nearest neighbor of x_i before the breakthrough at $x_k y_l$, we have the following inequality:

(8) $$d(x_i, y_j) - u_i - v_j \leqq d(x_i, y_q) - u_i - v_q.$$

Let us now consider the convex quadrilateral whose vertices are x_i, $y_{l(s)}$, y_j, and $x_{k(s)}$ (in clockwise order). The triangle inequality implies that $D(x_{k(s)}, y_{l(s)}) + D(x_i, y_j) \geqq D(x_{k(s)}, y_j) + D(x_i, y_{l(s)})$. By Lemma 6, the right-hand side of this inequality is nonnegative. Thus we obtain the following sequence of inequalities:

$$d(x_{k(s)}, y_{l(s)}) - u_{k(s)} - v_{l(s)} + d(x_i, y_j) - u_i - v_j \geqq 0,$$

$$-\delta' + d(x_i, y_j) - u_i - v_j \geqq 0 \quad \text{by (6)},$$

$$-\delta' + d(x_i, y_q) - u_i - v_q \geqq 0 \quad \text{by (8)},$$

$$d(x_i, y_q) - u'_i - v_q \geqq 0 \quad \text{by (7)},$$

$$d(x_i, y_q) - u^*_i - v^*_q \geqq 0 \quad \text{by (5) and the fact that } v^*_q = v_q.$$

The last inequality shows that the weighted distance between x_i and any remaining point of R is nonnegative. This completes the proof. □

The algorithm that we outlined at the beginning of this section can now be described in detail. Since the algorithm is recursive, it is presented under the guise of a recursive procedure. In the body of the procedure Find_Matching (Fig. 4), we use the same notation as in §§ 3 and 4.

Observe that the structure of this algorithm is similar to that of an algorithm for merging two sorted lists. This is not surprising, since the algorithm must "process" the set of negative edges of $G_w(P)$ in its total order, and by Lemma 6, the latter set is the union of the sets of negative edges of G_1 and G_2. While merging the two lists (one for G_1 and one for G_2), the algorithm discards the nonnegative edges and updates matching M when it discovers a negative one. The Boolean variable found_type1 is assigned the value "true" if the current edge of G_1 is a critical edge, and the value "false" otherwise. The variable found_type2 plays a similar role for G_2. Note that the value "true" cannot be assigned to both found_type1 and found_type2. If either one has the value "true," the instructions of lines 16–18 or 20–22 update M, Q, δ, and $List_1$ or $List_2$.

Note that in the algorithm, the meaning of δ is slightly different from the meaning ascribed to it at the beginning of this section. Here δ is positive when the previous critical edge belongs to G_1, and negative when it belongs to G_2. Thus the revised distance associated with x_iy_j (where x_iy_j belongs to G_1) is $D(x_i, y_j) + \delta$, while the revised distance associated with x_py_q (where x_py_q belongs to G_2) is $D(x_p, y_q) - \delta$ (see the update lemma).

Finally, we treat the edge x_0y_{r-1} in the same fashion as any other edge of G_1. We observe that if x_0y_{r-1} belongs to G_1, then it is the last edge scanned by the procedure. If it is also a negative edge, the edges of $\{y_0x_1, y_1x_2, \cdots, y_{r-2}x_{r-1}\}$ are added to M at line 16 and all the points of Q (with the exception of x_0 and y_{r-1}) are removed from Q. Since x_0y_{r-1} itself is added to M at line 24, we have added the edges y_0x_1, $y_1x_2, \cdots, y_{r-2}x_{r-1}, y_{r-1}x_0$ to M. The correctness of this step follows from the optimality

procedure Find_Matching (P: set of points; **var** M: matching);
begin
1. partition P into $P_1 = \{x_0, y_0, \cdots, x_{\lfloor n/2 \rfloor}, y_{\lfloor n/2 \rfloor}\}$ and $P_2 = \{x_{\lfloor n/2 \rfloor+1}, y_{\lfloor n/2 \rfloor+1}, \cdots, x_{n-1}, y_{n-1}\}$;
2. Find_Matching (P_1, N_1); Find_Matching (P_2, N_2);
3. $M \leftarrow$ {the matching edges that belong to the extensible sets found at Step 2};
4. $Q \leftarrow P \backslash$ {the endpoints of the edges of M};
5. compute G_1 and G_2 for Q;
6. let $List_1$ (respectively, $List_2$) be the list of all edges of G_1 (respectively, G_2) in their total order;
7. $\delta \leftarrow 0$;

repeat {*Loop Invariant*: the union of M and an optimal matching of Q is an optimal matching of P.}
8. **let** $x_i y_j$ (respectively, $x_p y_q$) be the first edge of $List_1$ (respectively, $List_2$) if it exists;
9. **if** ($List_1$ is not empty) **and** ($List_2$ is not empty) **and** ($x_i y_j$ and $x_p y_q$ intersect)
10. **then** found_type1 := $(D(x_i, y_j) + \delta < 0)$; found_type2 := $(D(x_p, y_q) - \delta < 0)$;
11. remove $x_i y_j$ from $List_1$; remove $x_p y_q$ from $List_2$
12. **elseif** ($List_2$ is empty) **or** (($List_1$ is not empty) **and** ($x_i y_j$ precedes $x_p y_q$ in the total order))
13. **then** found_type1 := $(D(x_i, y_j) + \delta < 0)$; found_type2 := false; remove $x_i y_j$ from $List_1$
14. **else** found_type2 := $(D(x_p, y_q) - \delta < 0)$; found_type1 := false; remove $x_p y_q$ from $List_2$

end;
15. **if** found_type1

then {$x_i y_j$ is the critical edge.}
16. $M \leftarrow M \cup \{y_i x_{i+1}, y_{i+1} x_{i+2}, \cdots, y_{i-1} x_j\}$; $Q \leftarrow Q \backslash \{y_i, x_{i+1}, y_{i+1}, \cdots, y_{j-1}, x_j\}$;
17. remove from $List_2$ all the edges of the form $x_r y_s$, where $r \leqq j$ or $s \geqq i$;
18. $\delta \leftarrow u_i + v_j - d(x_i, y_j)$

end;
19. **if** found_type2

then {$x_p y_q$ is the critical edge.}
20. $M \leftarrow M \cup \{x_{q+1} y_{q+1}, \cdots, x_{p-1} y_{p-1}\}$; $Q \leftarrow Q \backslash \{x_{q+1}, y_{q+1}, \cdots, x_{p-1}, y_{p-1}\}$;
21. remove from $List_1$ all the edges of the form $x_r y_s$, where $r > q$ or $s < p$;
22. $\delta \leftarrow -u_p - v_q + d(x_p, y_q)$

end
23. **until** ($List_1$ is empty) **and** ($List_2$ is empty);
24. $M \leftarrow M \cup \{x_0 y_0, x_1 y_1, \cdots, x_{r-1} y_{r-1}\}$ (where $\{x_0, y_0, x_1, y_1, \cdots, x_{r-1}, y_{r-1}\}$ is the current value of Q.)

end Find_Matching;

FIG. 4. *Algorithm Find_Matching.*

lemma. On the other hand, if the last edge scanned by the procedure is not a negative edge, the assignment of line 24 will assign a correct value to M (again by the optimality lemma).

THEOREM 1. *Let P be a set of $2n$ points lying on the boundary of a convex polygon. Algorithm Find_Matching computes a minimum-weight matching of P in time $O(n \log(n))$ and space $O(n)$.*

Proof. To prove the correctness of the algorithm, we show that the loop invariant is correct. The invariant obviously holds before the execution of the first iteration, since the set of matching edges assigned to M at line 3 is extensible. By Lemma 6, the first negative edge scanned by the algorithm is a critical edge, and by Lemma 5, M is an extensible matching after the execution of line 16 or line 20. On the other hand, lines 11, 17, and 21 update the weighted-nearest neighbor graphs correctly (cf. the retaining lemma and the discarding lemma). Finally, lines 18 and 22 update δ correctly as shown by the update lemma. Hence the loop invariant is correct. As we have observed in the paragraph preceding Theorem 1, the assignment of line 24 will assign to M the value of an optimal matching of P. This completes the proof of correctness.

It is obvious that the execution time of the **repeat** loop is proportional to the lengths of $List_1$ and $List_2$, which contain at most n edges. It remains to show that G_1 and G_2 can be computed in linear time. Consider a matrix A such that A_{ij} is the

weighted distance between x_i and y_j for $x_i \in R$ and $y_j \in L$. Note that the entry A_{ij} can be evaluated in a constant time. This matrix is easily proved to be *totally monotone*, as defined in Aggarwal et al. [1]. (The matrix A is totally monotone if for all i, j, k, l such that $i<k$ and $j<l$, $A(i,j) \geqq A(i,l)$ implies that $A(k,j) \geqq A(k,l)$.) We observe that to compute G_1, we need only find a minimum entry in each of the rows of A. Aggarwal et al. [1] give a linear algorithm for finding the row-minima of a totally monotone matrix, which allows us to compute G_1, and also G_2, in time $O(|L|+|R|)$.

Thus the conquer phase of our algorithm takes $O(n)$ time, and we obtain a familiar recurrence relation for $T(n)$, the time complexity of our algorithm:

$$T(n) \leqq 2T(n/2) + cn.$$

Thus, $T(n)$ is $O(n \log(n))$. The space requirement of our algorithm is easily seen to be $O(n)$. This completes the proof of the theorem. □

5. The matching problem for simple nonconvex polygons. In this section we generalize the result of § 4 by considering simple nonconvex polygons. Suppose that P is a set of $2n$ points lying on the boundary of a simple nonconvex polygon; we assume that the vertices of the polygon are included in P. We use P to denote the polygon as well. We assign weights to edges as follows: the weight of an edge equals the Euclidean distance between its endpoints if the endpoints see each other in the polygon, and $+\infty$ otherwise. As before, the weight of a matching is the sum of the weights of all the pairs (edges) in the matching.

We obtain an equivalent but more useful formulation of the minimum-weight matching problem by using the (non-Euclidean) shortest-path metric. Given two points x and y, let $g(x, y)$ denote a shortest path that joins x and y without leaving the polygon. We define the *shortest-path distance* between x and y to be the length of $g(x, y)$ and denote it by $|g(x, y)|$. The *generalized weight* of a pair $\{x, y\}$ is the shortest-path distance between x and y, and the *generalized weight* of a matching is the sum of the generalized weights of all the pairs in it. Then we have the following lemma.

LEMMA 10. *Let M be a matching whose generalized weight is minimum, and let a and b be two points such that $\{a, b\} \in M$. Then a and b can see each other in the polygon.*

Proof. Let us assume that a and b cannot see each other in the polygon. Then the shortest path $g(a, b)$ contains at least a vertex of P in its interior. Let $x \in P$ be such a vertex, and let y be the point of P such that $\{x, y\}$ belongs to M. If $x \notin g(a, y)$, then a straightforward calculation shows that $|g(a, b)|+|g(x, y)|>|g(a, y)|+|g(x, b)|$. Otherwise $x \notin g(b, y)$, and a similar calculation shows that $|g(a, b)|+|g(x, y)|>|g(a, x)|+|g(y, b)|$. In the former case the matching $(M \backslash \{ab, xy\}) \cup \{ay, xb\}$ has a smaller weight than M (in the generalized sense), and in the latter case, $(M \backslash \{ab, xy\}) \cup \{ax, yb\}$ has a smaller weight than M. In either case we have a contradiction and the proof is completed. □

Lemma 10 allows us to use the algorithm of the preceding section to find a minimum-weight matching of P, where we use the shortest-path metric instead of the Euclidean metric. The correctness of the algorithm is easily established, but the time complexity grows by a factor of $\log n$ because of the shortest-path distance computations. While the Euclidean distance between two points is easily calculated in $O(1)$ time, finding the shortest-path distance between two points of a nonconvex polygon is far from trivial. We use a data structure due to Guibas and Hershberger [5], which after $O(n \log(n))$ time preprocessing allows the shortest-path distance between two points to be computed in $O(\log(n))$ time. This leads to the following theorem.

THEOREM 2. *Let P be a set of $2n$ points on the boundary of a simple nonconvex polygon, where we assume that the vertices of the polygon are included in P. Let the weight of an edge equal the Euclidean distance between its endpoints if the endpoints see each other in the polygon, and $+\infty$ otherwise. Then the algorithm Find_Matching computes a minimum-weight matching of P in time $O(n \log^2(n))$ and space $O(n)$.* □

6. The assignment problem. In this section we consider the bipartite version of the matching problem. Let R and B be two sets of n points lying on the boundary of a convex polygon. We let P denote the set $R \cup B$. A *bipartite matching* (or assignment) is a partition of the points of P into n pairs of the form rb, where $r \in R$ and $b \in B$. The weight (or cost) of an assignment is the sum of the weights of all the edges contained in it. (As before, the weight of edge rb is the Euclidean distance $d(r, b)$.) In the rest of this section, we describe an $O(n \log(n))$ time algorithm for computing a minimum-weight assignment of $R \cup B$.

Let $z_0, z_1, \cdots, z_{2n-1}$ be a labeling of the points of P in counterclockwise order. We first consider a special case of the assignment problem, namely, the case where z_{2i} belongs to R and z_{2i+1} to B for $i = 0, 1, \cdots, n-1$. In other words, the even points are "red" and the odd points "blue." Call this special case the *basic problem.* We observe that the basic problem is almost identical to the "monochromatic" matching problem solved in § 4. This is so because an assignment of P is also a matching of P, and an optimal matching of P consists of edges of the form z_iz_j, where i is even and j odd. Hence to solve the basic problem, we can ignore the color classes and apply Algorithm Find_Matching to compute a minimum-weight assignment.

Next we demonstrate that an arbitrary assignment problem can be reduced to a collection of basic problems. Let z_p and z_q be two points of P such that $p < q$. We let $NR(z_p, z_q)$ (respectively, $NB(z_p, z_q)$) denote the number of red (respectively, blue) points lying between z_p and z_q, but not including z_p and z_q, in a counterclockwise traversal of the boundary. Since the edges of a minimum-weight assignment do not cross, the following lemma is straightforward.

LEMMA 11. *Let R and B be two sets of points lying on the boundary of a convex polygon, where $|R| = |B| = n$. Then any optimal assignment of $R \cup B$ consists of edges of the form z_iz_j, where $i < j$, such that $NR(z_i, z_j) - NB(z_i, z_j) = 0$.* □

Without loss of generality, assume that z_0 is a red point. By Lemma 11, if z_0z_j belongs to an optimal assignment of P, then $NR(z_0, z_j) - NB(z_0, z_j) = 0$. This motivates the following definition. A blue point z_j is said to be of *discrepancy* k (i.e., disc $(z_j) = k$) if $NR(z_0, z_j) - NB(z_0, z_j) = k$. Similarly, a red point z_i, for $z_i \neq z_0$, is said to be of *discrepancy* k if $NR(z_0, z_i) - NB(z_0, z_i) = k - 1$; z_0 itself is of discrepancy zero. The following lemma shows that z_iz_j may belong to an optimal assignment of P only if z_i and z_j have the same discrepancy.

LEMMA 12. *Let R and B be defined as in Lemma 11. Let z_i and z_j be two points such that $i < j$ and z_i and z_j have different colors. Then $NR(z_i, z_j) - NB(z_i, z_j) = 0$ if and only if* disc $(z_i) =$ disc (z_j).

Proof. Let us first assume that disc $(z_i) =$ disc $(z_j) = k$. There are two cases to consider: the case where z_i is red, and the case where z_i is blue. We shall treat the first case only, since the other one is similar. If $z_i = z_0$, then the lemma follows from the definition of discrepancy. Otherwise, z_i lies between z_0 and z_j in a counterclockwise traversal of the boundary. We have the following equalities:

$$NR(z_i, z_j) = NR(z_0, z_j) - NR(z_0, z_i) - 1,$$

$$NB(z_i, z_j) = NB(z_0, z_j) - NB(z_0, z_i),$$

$$
\begin{aligned}
NR(z_i, z_j) - NB(z_i, z_j) &= \{NR(z_0, z_j) - NR(z_0, z_i) - 1\} - \{NB(z_0, z_j) - NB(z_0, z_i)\} \\
&= \{NR(z_0, z_j) - NB(z_0, z_j)\} - \{NR(z_0, z_i) - NB(z_0, z_i)\} - 1 \\
&= k - (k-1) - 1 \\
&= 0 \quad \text{by the definition of discrepancy.}
\end{aligned}
$$

Conversely, assume that $NR(z_i, z_j) - NB(z_i, z_j) = 0$ and disc $(z_i) = k$. We only treat the case where z_i is red, since the other case is similar. We have

$$
\begin{aligned}
NR(z_0, z_j) - NB(z_0, z_j) &= \{NR(z_0, z_i) + NR(z_i, z_j) + 1\} - \{NB(z_0, z_i) + NB(z_i, z_j)\} \\
&= NR(z_0, z_i) + 1 - NB(z_0, z_i) \quad \text{by assumption} \\
&= (k-1) + 1 \\
&= k \quad \text{by the definition of discrepancy.}
\end{aligned}
$$

Thus disc $(z_j) = k$. This completes the proof of the lemma. □

We are now ready to show that our assignment problem can be decomposed into a collection of basic problems. Let P_k (for $k = -n+1, -n+2, \cdots, n-1$) denote the set of points of discrepancy k, and let M_k be an optimal assignment of P_k. Then Lemmas 11 and 12 imply that $\bigcup_{k=-n+1}^{n-1} M_k$ is an optimal assignment of P. The following lemma shows that the assignment problem on P_k (for $k = -n+1, -n+2, \cdots, n-1$) is a basic problem.

LEMMA 13. *Let P_k (for $k = -n+1, -n+2, \cdots, n-1$) be the subset of P defined above. Then the problem of finding an optimal assignment of P_k is a basic problem.*

Proof. Observe that there cannot be two consecutive *red* or two consecutive *blue* points on the boundary of P_k, since that would contradict the assumption that any two points of P_k have the same discrepancy. Therefore red and blue points alternate on the boundary of P_k. □

THEOREM 3. *Let R and B be two sets of points lying on the boundary of a convex polygon, where $|R| = |B| = n$. Then an optimal assignment of $R \cup B$ can be computed in time $O(n \log(n))$ and space $O(n)$.*

Proof. We note that it is possible to partition the points of P into discrepancy classes in $O(n)$ time; a single counterclockwise traversal of the boundary suffices. Once the P_k (for $k = -n+1, -n+2, \cdots, n-1$) have been computed, we apply Theorem 1 to each of the subproblems. (Of course, some discrepancy classes may be empty.) Algorithm Find_Matching takes time $O(n_k \log(n_k))$ and space $O(n_k)$ to solve problem P_k, where $n_k = |P_k|$. Thus the algorithm sketched above takes $O(n \log(n))$ time and $O(n)$ space to solve the assignment problem. □

To conclude this section, we note that the algorithm for solving the assignment problem can be easily adapted to the case of a simple nonconvex polygon, where we define the weight of a pair $\{a, b\}$ to be the length of a shortest path between a and b.

THEOREM 4. *Let R and B be two sets of points lying on the boundary of a simple nonconvex polygon, where $|R| = |B| = n$. Then an optimal assignment of $R \cup B$ with respect to the shortest-path metric can be computed in time $O(n \log^2(n))$ and space $O(n)$.* □

7. Maximum-weight matching. We now turn to the problem of computing a maximum-weight matching, where the weight of a matching is the generalized weight defined in § 5. Given a set P of $2n$ points on the boundary of a simple polygon, the weight of a pair $\{x, y\}$ is the shortest-path distance between x and y, and a maximum-weight matching of P is a partition of P into pairs such that the total weight of the

pairs is maximum. Surprisingly, computing a maximum-weight matching turns out to be much easier than computing a minimum-weight matching. Let a, b, c, and d be four points lying on the boundary of P. We say that the pairs $\{a, b\}$ and $\{c, d\}$ *cross* if the ordering of these points on the boundary is either a, c, b, d or a, d, b, c.

LEMMA 14 (crossing lemma). *In a maximum-weight matching of P, every two pairs cross.*

Proof. Let M be a matching that contains two noncrossing pairs, say, $\{a, b\}$ and $\{c, d\}$. Assume, without loss of generality, that the order of these points on the boundary is a, b, c, d. Elementary calculations show that

$$|g(a, c)|+|g(b, d)|>|g(a, b)|+|g(c, d)|.$$

Hence the matching $(M\backslash\{ab, cd\})\cup\{ac, bd\}$ has greater weight than M, and M is not a maximum-weight matching of P. □

The matching in which every two pairs cross is unique: if $z_0, z_1, \cdots, z_{2n-1}$ is an ordering of the points on the boundary, then the maximum-weight matching consists of the pairs z_iz_{i+n} for $i=0, 1, \cdots, n-1$. Computing the weight of the matching takes $O(n)$ time if the polygon is convex and $O(n \log(n))$ time otherwise. This proves the following theorem.

THEOREM 5. *Given a set of $2n$ points on the boundary of a simple polygon, we can compute its maximum-weight matching in $O(n)$ time if the polygon is convex and $O(n \log(n))$ time otherwise.*

8. Final remarks and open problems. The minimum-weight matching algorithm of Edmonds [4] is one of the classical results of combinatorial optimization. The complexity of Edmonds's algorithm stems in part from the fact that it can be applied to arbitrary graphs and edge-weights. In many practical situations, the graphs or the edge-weights have a special structure, and thus it is worthwhile to study special cases where more efficient algorithms can be designed. The geometric version of the matching problem is an obvious candidate for this study, but in spite of its apparent simplicity, the best algorithm known for this problem takes $O(n^{5/2} \log^4(n))$ time (cf. Vaidya [10]).

The results of this paper show that a natural restriction of the geometric matching problem leads to simple and efficient algorithms for this problem. A number of open problems are suggested by our work. The first one consists of improving the time bounds of our algorithms or showing that they are optimal.

It may also be possible to use the results of this paper to design a heuristic for the "general" version of the geometric matching problem. If M_0 is the weight of an optimal matching of S, P a simple polygon that spans the set S, and M_P the weight of an optimal matching of S where all the matching edges are constrained to remain in the polygon, then M_P/M_0 is the performance ratio of the heuristic for polygon P. What is the best ratio over all polygons P? How quickly can we find the polygon that minimizes this ratio?

Given a set of n points on a convex polygon, the traveling salesman cycle of this set of points coincides with their convex hull. A traveling salesman path through these points, however, is more difficult to compute. Using dynamic programming, we can obtain an $O(n^2)$ algorithm. Can the ideas of this paper be extended in order to improve this bound?

Finally, we might consider the matching problem for the vertices of a convex polytope in three dimensions. Can we solve this problem in, say, $O(n^2)$ time?

Acknowledgments. The authors thank Daniel Bienstock, Fan Chung, Nate Dean, Clyde Monma, and Avi Wigderson for many pleasant discussions during work on this

paper. They are especially thankful to James Park for pointing out an error in the previous version of the optimality lemma. The first author also expresses her thanks to Bellcore for their hospitality, in particular, the Division of Mathematics, Information Sciences, and Operations Research.

REFERENCES

[1] A. AGGARWAL, M. KLAWE, S. MORAN, P. SHOR, AND R. WILBER, *Geometric applications of a matrix-searching algorithm*, Algorithmica, 2 (1987), pp. 209-233.

[2] S. AKL, *A note on Euclidean matchings, triangulations and spanning trees*, J. Combin. Inform. and System Sci., 8 (1983), pp. 169-174.

[3] N. CHRISTOFIDES, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Tech. Report, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[4] J. EDMONDS, *Maximum matching and a polyhedron with* (0, 1) *vertices*, J. Res. Nat. Bur. Standards, 69B (1965), pp. 125-130.

[5] L. GUIBAS AND J. HERSHBERGER, *Optimal shortest path queries in a simple polygon*, in Proc. 3rd Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1987, pp. 50-63.

[6] R. KOPPE, *Automatische Abbildung eines planaren Graphen in die Ebene mit beliebig vorgebbaren Orten der Knotenbilder*, Computing, 20 (1978), pp. 61-73.

[7] E. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[8] M. SHAMOS, *Computational geometry*, Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.

[9] P. VAIDYA, *Geometry helps in matching*, SIAM J. Comput., 18 (1989), pp. 1201-1225.

[10] M. WERMAN, S. PELEG, R. MELTER, AND T. KONG, *Bipartite graph matching for points on a line or a circle*, J. Algorithms, 7 (1986), pp. 277-284.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 423-450, June 1991
© 1991 Society for Industrial and Applied Mathematics
002

THE COMMUNICATION COMPLEXITY OF ATOMIC COMMITMENT AND OF GOSSIPING*

OURI WOLFSON† AND ADRIAN SEGALL‡

Abstract. The problem of atomic commitment of a transaction in a distributed database is considered. This is a variant of the famous gossiping problem. Given a set of communication costs between pairs of participant sites, it is established that the necessary communication cost for any atomic commitment algorithm is twice the cost of a certain minimum spanning tree. This paper establishes the necessary communication time for any atomic commitment algorithm, given a set of communication delays between pairs of participant sites, and the time at which each participant completes its subtransaction. Then, it is determined that both lower bounds are also upper bounds in the following sense. There is an efficient (i.e., polynomial-time) algorithm that, in the absence of failures, has a minimum communication cost. There is another efficient algorithm that, in the absence of failures, has a minimum communication time. However, unless P = NP, there is no efficient algorithm which has a minimum communication complexity, namely, one for which the product of communication cost and communication time is minimum. Next, a simple, linear time, distributed algorithm, called TREE-COMMIT, whose communication complexity is not worse than p times the minimum complexity, where p is the number of participants, is presented. Finally, it is demonstrated that TREE-COMMIT is superior to the existing variants of the two-phase commit protocol.

Key words. commit protocols, database consistency, distributed databases, optimal protocols, transaction management, gossiping

AMS(MOS) subject classifications. 68P15, 68P20, 68M10, 68Q25

1. Introduction.

1.1. The problem. In a distributed database, a transaction consists of several subtransactions, each running at a different site that has a local database. When the subtransaction at a site completes, i.e., the corresponding process finishes, the local database manager knows whether it completed successfully or unsuccessfully.[1] The *atomic commitment* problem (see [G]) for the set of local database managers is to determine the decision of each manager concerning the changes made by the transaction to the local database. The manager may either validate these changes (commit the subtransaction) or invalidate them (abort the subtransaction). The generally accepted solution to the atomic commitment problem is the following: if all the completions are successful, then commit all the subtransactions (yielding a committed transaction), otherwise abort all the subtransactions (yielding an aborted transaction). To achieve atomic commitment, the local database managers execute a distributed algorithm, exchanging "yes" and "no" votes. A "yes" vote indicates the successful completion of a subtransaction, and a "no" vote indicates an unsuccessful completion.[2]

In this paper, we first consider the load that atomic commitment algorithms place on the communication network, in the following sense. Each subtransaction of a given

---

* Received by the editors December 28, 1988; accepted for publication (in revised form) May 1, 1990.

† Department of Computer Science, Columbia University, New York, New York 10027. The research of this author was supported by Defense Advanced Research Projects Agency grant #F-29601-87-C-0074; by the Center for Advanced Technology at Columbia University grants NYSSTF-CAT(89)-5 and CU01207901; and by National Science Foundation grant IRI-90-03341.

‡ Department of Computer Science, Technion-Israel Institute of Technology, Haifa 32000, Israel. The research of this author was supported by the Technion Fund for the Promotion of Research.

[1] For example, a subtransaction may complete unsuccessfully as a result of deadlock.

[2] Transactions are used in distributed systems other than databases, such as Argus ([LHJLSW]) and Camelot ([Sp]). The atomic commitment problem, as described, arises in such systems as well.

transaction runs at some node, called a *participant* site for that transaction. Additionally, with each pair of participants, i and j, is associated the cost, c_{ij}, of sending a message between i and j (in either direction). The costs may differ from one pair of participants to another. For example, the communication cost for a pair of participants may represent the distance in the communication network between the participants. The communication load that an execution of an algorithm places on the communication network is quantified in terms of its *communication cost*, i.e., the total cost of messages that the algorithm sends among the participants.

Traditionally, the communication load has been quantified in terms of the number of logical messages among participant sites, namely, *intersite messages*. (Implicitly, c_{ij} was taken to be one for every pair of participants, i and j.) However, this may be too coarse a measure for comparing the load that different atomic commitment algorithms place on the network. To demonstrate this, next we shall present three commitment executions, or instances, for a transaction. All of them use the same number of intersite messages but put different loads on the communication network, since they use different numbers of *network messages*, i.e., messages between neighbors in the communication network (regardless of whether they are participants or nonparticipants).

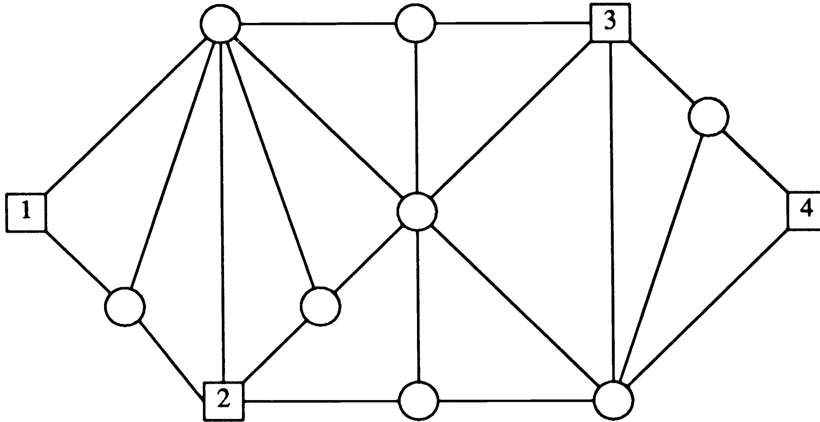


FIG. 1.1

For example, assume that the transaction executes in the computer-communication network given in Fig. 1.1. Squares represent participant sites, circles represent nonparticipant sites, and edges represent two-way communication links. A possible commitment instance, based on the "central site" algorithm of Lampson ([La]), is illustrated in Fig. 1.2(a): each one of the participants 2, 3, and 4 sends its "yes" vote directly to participant 1, which also votes "yes," commits, and sends the commit decision separ-

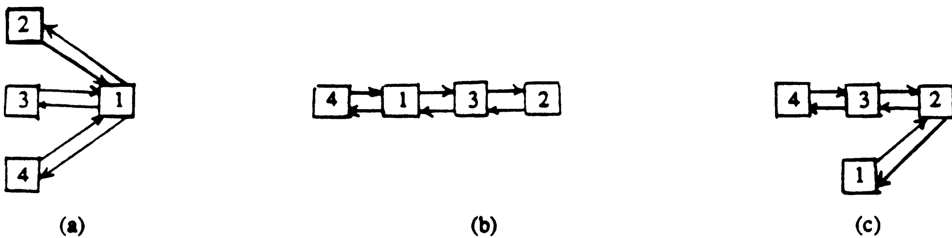


FIG. 1.2

ately to each of participants 2, 3, and 4 (each of which commits when it receives the message). This instance uses six intersite messages. Another possible instance, based on the "linear" algorithm of Gray ([G]), is illustrated in Fig. 1.2(b): 4 sends its "yes" vote to 1, which then sends its and 4's "yes" votes to 3, which then sends its and 1's and 4's "yes" votes to 2; 2 then votes "yes," commits, and sends the commit decision to 3, which sends it to 1, which sends it to 4. This instance also uses six intersite messages. In fact, from the lower bound result of Dwork and Skeen [DS1] for atomic commitment, we know that any four-participant atomic commitment execution requires at least six intersite messages. Finally, consider the following instance that also uses six messages (see Fig. 1.2(c)): 4 sends a "yes" vote to 3, which sends its and 4's "yes" votes to 2; meanwhile, 1 also sends a "yes" vote to 2, which, after receiving the two messages, also votes "yes" and commits, after which the commit messages travel from 2 to 3 to 4 and from 2 to 1. If we assume that the intersite messages travel through the shortest path in the network, then the first instance takes 18 network messages, the second takes 20, and the latter takes only 14. Since the actual communication load on the network is reflected by the number of network messages, the third instance above imposes the least load on the network.

The other measure that we consider in comparing the performance of commit protocols is the *communication time*. Communication time of an instance is defined as the interval of time starting when the first subtransaction completes, and ending when the last participant commits its subtransaction. In this respect, we also depart from traditional models (e.g., [DS1], [R]), which for the purpose of analysis, assume a synchronous communication network and simultaneous completion of all subtransactions. The synchronous communication implies, in particular, a unit-time intersite message delay, independent of the network location of the sender and receiver.

In our model we allow, more realistically, arbitrary subtransaction completion times as well as different intersite delays. Particularly, different participants may complete their subtransaction at different times, and intersite message delays may differ from one sender–receiver pair to another. Therefore, we dispose of unrealistic assumptions regarding synchronous operation of geographically dispersed processors.

Finally, let us mention a different formulation of the problem addressed. There is a large body of research on "gossiping," and many variants of this problem were studied in the literature (see [HHL] for a survey). Our model and our results apply to the following variant of the gossiping problem; it has not been solved before, and actually, our necessary and sufficient cost results solve an open problem discussed by Cot ([C]). There is a set, A, of individuals, each of which knows a unique piece of a gossip. Additionally, for each pair of individuals i and j, there is a cost, c_{ij}, and a travel-time (corresponding to the intersite delay in atomic commitment terminology), t_{ij}, associated with a letter (or a message) from i to j. Also, individual i learns its own piece of the gossip at time τ_i (corresponding to the subtransaction completion time). A solution consists of a partially ordered set of pairs of individuals. Each pair represents a letter sent from the first to the second, and the partial order represents the relative times at which the letters are sent. By assumption, each letter contains all the pieces of the gossip known to the sender when sending the letter. At the end, each individual must know the whole gossip. In § 8.2, we discuss another unsolved variant of gossiping, and the implications of the results in this paper to that variant.

The atomic commitment problem is a variant of gossiping, in which the unique pieces of the gossip are the votes, and after collecting all the votes, each individual computes their conjunction. Although throughout the paper we use the atomic commitment terminology, it is clear that the issues addressed are independent of the function

computed by each processor after the vote collection. In particular, a commitment instance is also a gossiping instance. So, for example, assume that each one of the individuals 1, 2, 3, and 4, knows a unique piece of a gossip. Then the instance in Fig. 1.2(a) represents the sending of all the pieces to 1, which then transmits the whole gossip to the other individuals. Therefore, all our results carry over to the general "gossiping" case, with one exception. On some occasions, we briefly discuss the abort case of a distributed transaction. In this case a participant does not necessarily have to collect all the votes; when receiving a "no" vote it already knows the result of the conjunction. Thus, the abort case of atomic commitment does not have a gossiping counterpart, making atomic commitment a variant, rather than a special case, of gossiping.

1.2. Our results. First we establish the necessary communication cost of atomic commitment, namely the lower bound on the communication cost of any atomic commitment algorithm, for a given a set of participants, and an associated set of communication costs between every pair of participants. The necessary communication cost is twice the weight of a minimum spanning tree in a complete graph, called the cost graph. It has the participants as nodes, and the costs as the weights of the edges. For example, in Fig. 1.3 we show the cost graph of the participant sites and network of Fig. 1.1, assuming that the communication cost between every pair of participants is the distance between them in the network. We shall explain at the beginning of § 3 that this result would have been trivial had we known that there is an atomic commitment execution of minimum communication cost which has a single coordinator (a participant to which all other participants send their votes). However, we do not know this a priori, and, in fact, the most difficult part of the lower bound proof is showing that there always exists a minimum communication cost instance with one coordinator.

Then, we show that the necessary cost is also sufficient for atomic commitment by presenting an algorithm that achieves the lower bound. Subsequently, we establish the necessary communication time for atomic commitment. Then we demonstrate that it is also sufficient, by demonstrating that the decentralized algorithm (introduced previously by Skeen ([Sk]), and explained briefly in the next subsection) achieves the

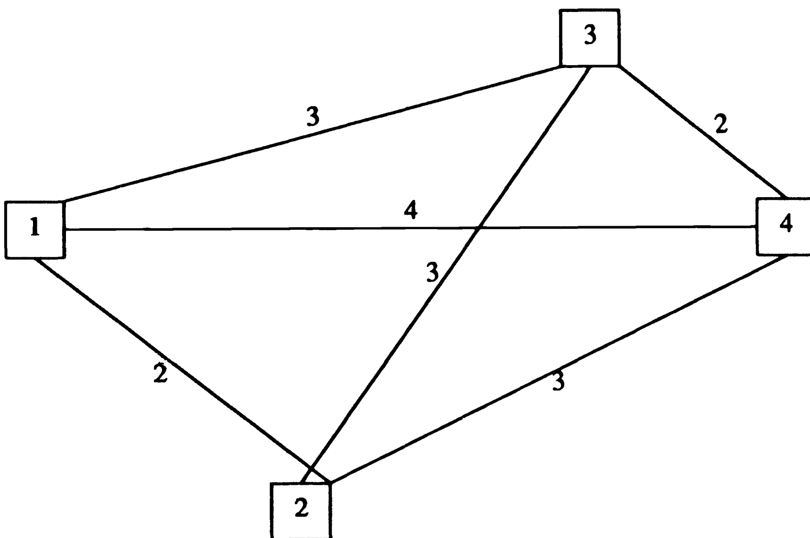


FIG. 1.3

communication time lower bound. We define the communication complexity of an instance as the product of its communication time and its communication cost, and we point out that it is NP-complete to find the instance of minimum communication complexity. However, we propose the algorithm TREE-COMMIT, in which messages propagate along the edges of a tree spanning the participants. For a minimum spanning tree of the cost graph, TREE-COMMIT has minimum communication cost. Its communication complexity (communication time) can be worse than the minimum communication complexity (minimum communication time) by a factor that is bounded by the number of participants, p. The computation time (as opposed to the communication time) of each one of the p participants executing TREE-COMMIT is linear in the size of the input. This implies that TREE-COMMIT is a distributed variant of a polynomial-time approximation algorithm, for the NP-complete problem of finding the minimum communication complexity instance, and that the approximation has a bound, p, on the error. Moreover, TREE-COMMIT achieves the bound on the communication complexity, without the participants knowing the subtransaction completion times, or the communication delays. Then we analyze the decentralized algorithm, which is communication time optimal. We show that its communication complexity (cost) can be worse than the minimum communication complexity (cost) by a factor of order p^2.

The decentralized algorithm is a variant of the two-phase-commit paradigm. Two other well-known variants are the ones mentioned above, namely the central-site and the linear algorithms (they are briefly explained in the next subsection). The algorithm introduced in this paper, TREE-COMMIT, is superior to these two variants in a very strong sense. It can be adapted, by varying its communication tree, to have exactly the same communication cost as the central-site algorithm (alternatively, it can be adapted to have the same communication cost as the linear algorithm). Then, for any set of subtransaction completion times and intersite communication delays, its communication time cannot be higher than that of the central-site algorithm (the linear algorithm). Furthermore, there are cases, i.e., completion times and delays, for which it is half the communication time of the central-site algorithm (the linear algorithm).

In this paper we introduce a novel model of an atomic commitment instance. Although other models, such as finite state automata ([Sk]) and knowledge theoretic ([H2]), exist in the literature, we chose to represent the instance by a directed acyclic graph, representing the time-order of events and messages. It enables us to establish the communication cost and communication time lower bounds, and to analyze the cost and time of algorithms, all in the same formalism. The communication cost is the total length of the arcs, and the communication time is the length of the longest path.

Our results are restricted to the case in which no failures occur while the commitment protocol is executed, although the lower bounds obviously also hold in models that allow failures. We mainly analyze the case in which each participant votes to commit the transaction (and thus the transaction commits), for the following reasons. Successful commitment represents the more likely outcome for transactions in most database systems, and it also represents a worst-case scenario from the communication cost viewpoint (see Theorem 3).

1.3. Other related work. Atomic commitment is a variant of a fundamental problem in distributed systems, namely distributed consensus. Fischer presents a survey of the subject ([F]), and Dwork and Skeen devise an interesting taxonomy of consensus problems ([DS2]). Hadzilacos presents an illuminating discussion on the applicability of the consensus problem results to the atomic commitment problem ([H1]), particularly

the types of failures that are meaningful for each problem. In short, almost all existing research concentrates on the effects of failures on achieving consensus in general, and sometimes atomic commitment, in particular.

In contrast, in this paper we concentrate on performance issues. Mohan, Lindsay, and Obermack also discussed performance issues of commitment protocols ([MLO]); however, our work differs from theirs in two respects. First, their atomic commitment algorithms are more complicated, mainly because of a more complicated transaction model. The model allows for the fact that there may not be any participant that knows the identity of all the other participants in the transaction. Second, [MLO] has assumed a one-unit cost for each message, as have the other previous works.

Informal discussions of the performance of atomic commitment algorithms in the absence of failures also appear in [BHG], [CP], [G], and [Sk], mainly in the context of different two-phase-commit variants. One of the most popular is the *central-site* algorithm (see Fig. 1.2(a)); a participant is designated as the "protocol coordinator" and each other participant sends its vote to the coordinator. The coordinator makes the decision and sends the "commit" or "abort" message to all the other participants. Another two-phase-commit variant is the *decentralized* algorithm, in which each participant sends its vote to all the other participants. Based on the received messages each participant makes the "commit" or "abort" decision. We shall show that this algorithm minimizes the communication time. Finally, in the *linear* algorithm (Fig. 1.2(b)), all the participants are sequentially ordered. Each participant sends its vote to the next one in the sequence. The last participant is the protocol coordinator, which reverses the flow direction, by sending the decision message to its predecessor in the sequence. The linear and central-site algorithms require $2(p-1)$ intersite messages, where p is the number of participants. Dwork and Skeen have formally proven that the number of intersite messages required by any atomic commitment protocol, in the absence of failures, is $2(p-1)$ ([DS1]). By contrast, note again that in the present work we consider the total communication cost of intersite messages, rather than their number. In the special case in which the communication cost of every intersite message is one, our cost lower bound result matches the $2(p-1)$ result.

1.4. Paper organization. The rest of this paper is organized as follows. In § 2 we introduce our model of a commit instance. In § 3 we establish the necessary communication cost for the atomic commitment problem, and in § 4 we provide a complete characterization of the minimum communication cost commit instances. In § 5 we establish the minimum communication time of an instance, and in § 6 we present the TREE-COMMIT algorithm. In § 7 we analyze TREE-COMMIT and compare it with the other algorithms discussed in this paper. In § 8 we conclude, and discuss future work.

2. Commit instances. In this section we provide some key definitions. In particular, we formally introduce our novel model of a commit instance. Intuitively, the instance executed by an atomic commitment algorithm is represented by the temporal, and thus partial, order of events occurring at the participants. Let P be a set of participants. Formally, an *instance* on P is a directed acyclic graph, $I=(E, A)$ (see Fig. 2.1(a)). E is a set of nodes, called *events*, and A is a set of arcs (i.e., directed edges). Every event *occurs* at some participant, and all the events occurring at a participant are totally ordered in I. Each event represents zero or more consecutive receives (each of an intersite message) at the participant, without an intervening send, followed by zero or more consecutive sends, without an intervening receive. The first event occurring at a participant also represents the completion of the corresponding subtransaction. Every pair of consecutive events occurring at a participant are connected by an arc called

an *order arc* (since it represents the order in which the two events occur at the participant). The other arcs of A are called *messages.* A message is an arc from an event called the *send* of the message, to an event called the *receive* of the message, which occurs at a different participant from the send. Only the last event occurring at a participant may send zero messages, and only the first event occurring at a participant may receive zero messages. Thus, into every event, except possibly the first one occurring at each participant, enters at least one message, and from every event, except possibly the last one occurring at each participant, exits at least one message. If there is a path in I between events a and b, then we say that a *happens* before b (in the sense of Lamport [L]), and b *happens* after a. We assume that a message sent at an event a contains all votes that happened before a. Three possible instances at a set of three participants are illustrated in Fig. 2.1.

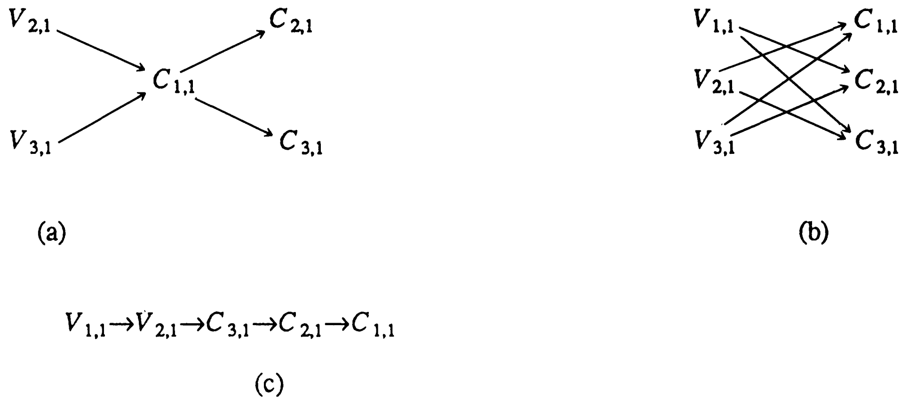


FIG. 2.1

Next, we comment on the representation of several message-receives and -sends by one event. For the purpose of this paper, the order of consecutive message-sends is irrelevant, as is the order of consecutive message-receives. For example, we do not distinguish between two "instances" in which the only difference is that at some participant the order of two consecutive receives is reversed. Only the relative order of blocks of receives and sends is relevant, since we assume that each message sent includes all, and only, the votes received before sending the message, and also includes the vote of the sender.

We assume that a participant may send messages only after its subtransaction completes. Consequently, the first event at each participant represents zero or more message-receives, followed by the corresponding subtransaction completion, followed by zero or more additional message-receives, followed by one or more message-sends.

A *commit* instance, I, is an instance that satisfies the following *commit requirement*: At each participant occurs at least one event, e, which happens after the first event occurring at every other participant. Any event such as e, that happens after some, and particularly the first, event at each other participant, is called a *C-event*; informally, when it occurs, its participant, say j, already knows the commit decision. The reason for this is that j has received all the "yes" votes from the other participants, and it knows that its own vote is "yes." The vote of some participant, say i, propagates to j along the path from the first event at i, to the C-event of j. The first C-event occurring at a participant, in addition to message sents and receives, also represents the validation of the changes made by the subtransaction, and the recording in stable storage of the

fact that the transaction is committed. A message sent by a C-event is called a *commit* message. Each message that is not a commit message is a "yes" *vote* message. An event which is not a C-event is called a *V-event.*

The motivation for the commit requirement is that each participant must receive the "yes" vote of every other participant, and vote "yes" itself, before knowing that it can commit ([DS1], [H2]). For example, the instances illustrated in Fig. 2.1 are commit instances. In our figures, the V-events (C-events) are denoted by a subscripted V (C). The label $V_{i,j}$ ($C_{i,j}$) of a node indicates that this is the jth V-event (C-event) occurring at participant i.

Notice that the commit requirement ensures that every commit instance is connected. Notice also that any event which happens after a C-event, must also be a C-event. Another observation is that a C-event may represent the receive of a "yes" vote message (for example, $C_{1,1}$ in Fig. 2.1(a)).

The variants of the two-phase commit protocol mentioned in the introduction are illustrated in terms of our model in Fig. 2.1. In order to prevent cluttering the figures we omit the order arcs; however, remember that any two consecutive events at a participant are connected by an order arc. An instance executed by the central-site algorithm, with participant 1 as the coordinator, is illustrated in Fig. 2.1(a). Instances of the decentralized and linear algorithms are illustrated in Figs. 2.1(b) and 2.1(c), respectively.

3. Minimum communication cost of commit instances. In this section we establish the minimum communication cost of a commit instance. This would have been quite straightforward, had we known that there exists a minimum communication cost commit instance in which all the votes are sent directly to one participant, the coordinator, which, after receiving all the votes, broadcasts the commit decision. This would have meant that the problem of atomic commitment at minimum communication cost can be decomposed into two problems, collect-at-minimum-cost and broadcast-at-minimum-cost, and these two problems can be solved independently. In Lemma 2 we show that there always exists a minimum communication cost commit instance that consists of a collect-to-one, followed by a broadcast, but we do not know this yet. Also, we do not know yet that there exists a minimum cost instance that has a minimum number of intersite messages, and therefore we cannot use this fact for the lower bound proof. At the end of the section we will present the FIXED-COORDINATOR algorithm, which achieves minimum communication cost, and consists of a collect-to-one, followed by a broadcast.

We start with some formal definitions. Let P be a set of participants. We suppose that P is associated with a set, c, of communication costs, consisting of a positive real number, c_{ij}, for each pair of different participants, i and j, in P. For each such pair, c_{ij} is the *communication cost* of a message from i to j, and we assume that it is equal to the cost of a message from j to i, namely, $c_{ij} = c_{ji}$. Given P and c, in every instance on P the communication cost of a message arc from (an event occurring at participant) i to (an event occurring at participant) j is c_{ij}; the communication cost of an order arc is taken to be zero. The communication cost of an instance I, denoted $cost(I)$, is the total communication cost of all the arcs in I. For example, if $c_{12} = c_{13} = 1$, then the cost of the instance on Fig. 2.1(a) is 4.

In this section we assume a fixed set of participants, P, and a fixed set of associated communication costs, c, and we establish the minimum communication cost of a commit instance on P. A commit instance of such cost will be referred to as a *minimum communication cost* instance. We denote by Ψ the set of commit instances on P. Given

a commit instance I, its C-subgraph, denoted C_I, is defined as the subgraph of I induced by its C-events.

LEMMA 1. *If I is a minimum communication cost instance of Ψ, then I has only one C-event at every participant. Furthermore, its C-subgraph is a forest of rooted trees.*

Proof. Assume that there are two or more C-events at some participant. Then, there must be at least one incoming message into the second C-event, by definition of a commit instance (in particular, that only the first event at a participant may represent zero receives). By omitting such a message, i.e., removing the arc corresponding to some message into the second C-event, a commit instance of strictly lower communication cost can be obtained.[3] The reason that the message can be omitted, is that the commit requirement is satisfied for the participant by the first C-event occurring at the participant. Consequently, there is only one C-event at every participant.

In order to show that C_I is a forest, it is now sufficient to show that there is no C-event having two incoming commit messages. But this fact is obvious; if there were such an event, then all but one of the incoming commit messages could be omitted to obtain an instance of lower communication cost. □

The single C-event at every participant, i, will be denoted by C_i. Assume that C_j is one of the roots in C_I, for some minimum communication cost instance, I. In such a case, we say that participant j is a *coordinator* of the instance I, and that C_j is a *boundary C-event* of I. Informally, this means that, in the execution I, participant j knows all the votes without receiving a commit message (that is, j knows all the votes without receiving a message from another participant that knew all the votes). Suppose now that I has two or more coordinators. Consider a V-event, V^0, which satisfies the following condition. It precedes at least two boundary C-events, say C_i and C_j, and, if V^0 has any V-event successors, then each one of them precedes only one boundary C-event (for example, V^0 is W^0 in Fig. 3.1, and the V-event successors are the W^i's $1 \leqq i \leqq n$). Such an event is called a *boundary* V-event. It is easy to see that every commit instance with two or more coordinators has a boundary V-event. Simply start at the first event at some participant, which by the commit requirement precedes every boundary C-event. Verify whether that event satisfies the above condition. If not, it means that one of its V-event successors precedes two or more boundary C-events. Repeat the verification at that successor, until a V-event with the following property is found: either it has no V-event successors, or each one of its V-event successors precedes only one boundary C-event.

Suppose that boundary V-event V^0 occurs at participant p_0. By Lemma 1 we know the structure of the C-subgraph of I, particularly that there is only one C-event that occurs at p_0, C_{p_0}. Let C_i be a boundary C-event that succeeds V^0, such that C_{p_0} is *not* in the tree of C_I rooted at the C_i. Since V^0 precedes more than one boundary C-event, there must be such a C-event. Then C_i is referred to as *associated* with boundary V-event V^0.

LEMMA 2. *There exists a minimum communication cost instance in Ψ, which has exactly one coordinator.*

Proof. Let I be a minimum communication cost instance, and suppose that it has two or more coordinators. We shall show that we can transform it into another commit

[3] If the omitted message was the only one exiting its send event, say e, and e is not the last event at its participant, then after the omission the instance has to be adjusted. Adjustment is by collapsing e and its consecutively following event at the participant. Two events e and f are collapsed by omitting the event f, and substituting e for f in the arcs of the instance (the arc (e, e) is omitted). A similar adjustment has to occur if the omitted message is the only one entering its receive event.

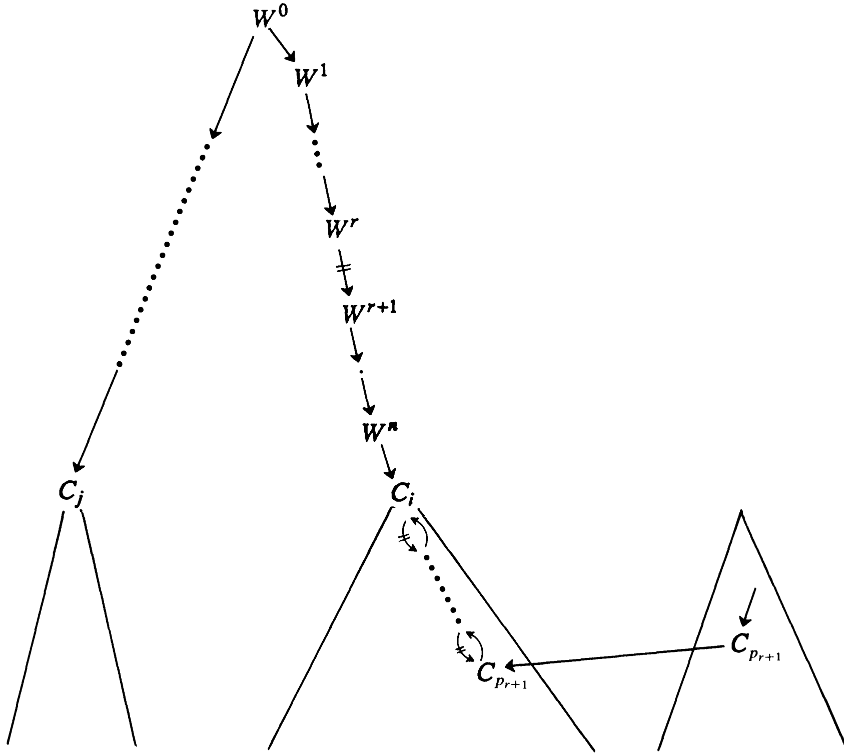


FIG. 3.1

instance, I', of equal communication cost, but with one less coordinator. The transformation replaces a vote message from some participant, i, to some participant, j, by a commit message from j to i. Since the costs of the two messages are equal, the costs of I and I' are equal. The effect of the transformation is to change the way in which one of the coordinators learns the votes of the other participants; that coordinator becomes a noncoordinator and learns the votes of the others via a chain of messages from one of the remaining coordinators. We now describe the selection of the vote message which will be replaced.

Suppose that V^0 is a boundary V-event that occurs at participant p_0, and let C_i be a boundary C-event *associated* with V^0. Denote a path from V^0 to C_i, by $V^0 = W^0, W^1, \cdots, W^n, W^{n+1} = C_i$, where $n \geqq 0$. Note that all events on the path, except the last one, are V-events. Let W^r, for $0 \leqq r \leqq n$, be the last event on this path for which the following condition is true: W^r occurs at a participant, p_r, for which C_{p_r} is in a tree different than the one rooted at C_i. Since W^0 satisfies the condition, there must be such a W^r. Denote by p_{r+1} the participant at which W^{r+1} occurs. By the definition of W^r, the event $C_{p_{r+1}}$ is the tree rooted at C_i. Now, to obtain a commit instance of minimum communication cost with one less coordinator, perform the following transformation:

(i) the arc $W^r \rightarrow W^{r+1}$ is replaced by a message from C_{p_r} to $C_{p_{r+1}}$,

(ii) the direction of the arcs on the path from C_i to $C_{p_{r+1}}$ is reversed, and

(iii) if transformations (i) and (ii) result in any event that consists of message-receives only, then that event and the one immediately following it at the same participant are collapsed into one event.

Figs. 3.1 and 3.2 provide two examples of the transformation. Fig. 3.1 illustrates the modifications performed on I, assuming that W^0 does have some V-event successors. Figs. 3.2(a) and 3.2(b) illustrate an instance, I, before and after the transformation, respectively; W^0 is V_2, and it has no V-event successors.

It is easy to see that the transformation results in a commit instance; denote it I'. The commit requirement is satisfied by I' because, by definition of W^0, the removal of the arc $W^r \to W^{r+1}$ can only disconnect paths to the C-events in the C-subgraph tree rooted at C_i. However, all those C-events are now preceded by $C_{p_{r+1}}$, which is in a different C-subgraph tree. Therefore, I' is a commit instance and has the same coordinators as I, except participant i, which is a coordinator in I, but is not a coordinator in I'. Moreover, the transformation performed on I does not alter the communication cost, hence I' has minimum communication cost.

To summarize, starting with a minimum communication cost instance, I, with two or more coordinators, we obtained a minimum communication cost instance, I', with one less coordinator. We can continue this procedure until a minimum communication cost instance with exactly one coordinator is obtained. □

Next, we will obtain an additional lemma; it will be used in § 4. In a minimum communication cost instance, I, having two or more coordinators, let V^0, p_0, and C_i, be as in the proof of Lemma 2. Namely, V^0 is a boundary V-event that occurs at participant p_0, and C_i is a boundary C-event that is associated with V^0. The fact that I is a minimum communication cost instance, implies that the path in I, denoted p, from V^0 to C_i, is unique, and consists of a single message arc, $V^0 \to C_i$. The reason for this is as follows. If this is not the case, then there are more messages on p, or there are additional paths from V^0 to C_i. As established in the proof of Lemma 2, one of the messages on p is from p_r to p_{r+1}. Then, the transformation in the proof of Lemma 2 can be augmented by the elimination of all messages on p, and on the other paths from V^0 to C_i. The resulting graph is a commit instance that has a cost strictly lower than I. This contradicts the fact that I has a minimum communication cost. Therefore, $V^0 \equiv W^r$, $W^{r+1} \equiv C_i$, $p_0 \equiv p_r$, $i \equiv p_{r+1}$, and the transformation in the proof of Lemma 2 simply replaces a message $V_{p_0,l} \to C_i$ by a message $C_{p_0} \to C_i$. We have therefore proved the following.

LEMMA 3. *In any minimum communication cost instance of* Ψ, *with two or more coordinators, there is a unique path from any boundary V-event,* V^0, *to any associated boundary C-event,* C_i, *and it consists only of the message* $V^0 \to C_i$. □

Now, let us define the *cost graph*, D, as the complete graph having the set of participants P as its nodes. The weight of each edge (i, j) equals c_{ij}. The cost of a minimum spanning tree in D is denoted by $CMST(c, P)$.

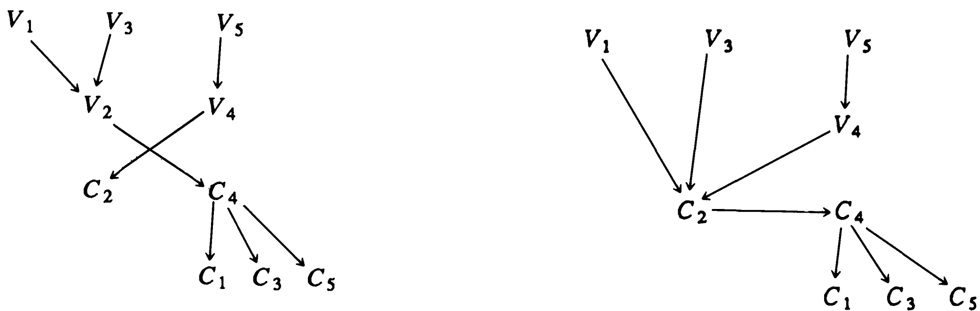


FIG. 3.2

Given T_1 and T_2, two not necessarily different spanning trees of D, we define a *commit instance on* T_1 *and* T_2 *coordinated at some participant* $k \in P$. It is denoted by $I(k, T_1, T_2)$, and specified as follows. Denote by T_1' the directed graph obtained from T_1 by directing its edges such that from every node there is a path to k (the graph obtained is called an *oriented* tree with sink k). Also, denote by T_2' the directed graph obtained from T_2 directing its edges to form a rooted tree, with root k.

One obtains the instance $I(k, T_1, T_2)$ as a result of the following modifications:

(1) relabel the nodes from T_1'; node i is relabeled V_i;

(2) relabel the nodes from T_2'; node i is relabeled C_i;

(3) node V_k is omitted, and the arcs entering it are modified to enter C_k instead;

(4) add the order arcs $V_i \to C_i$ for each i, except the coordinator.

Intuitively, $I(k, T_1, T_2)$ is a commit instance that has vote messages which correspond to the arcs of T_1', and commit messages which correspond to the arcs of T_2'. The definition of $I(k, T_1, T_2)$ is illustrated in Fig. 3.3. Given minimum spanning trees T_1 (a) and T_2 (b), the instance $I(4, T_1, T_2)$ is illustrated in (c). Clearly, if T_1 and T_2 are minimum spanning trees, then the communication cost of $I(k, T_1, T_2)$ is $2 \cdot CMST(c, P)$.

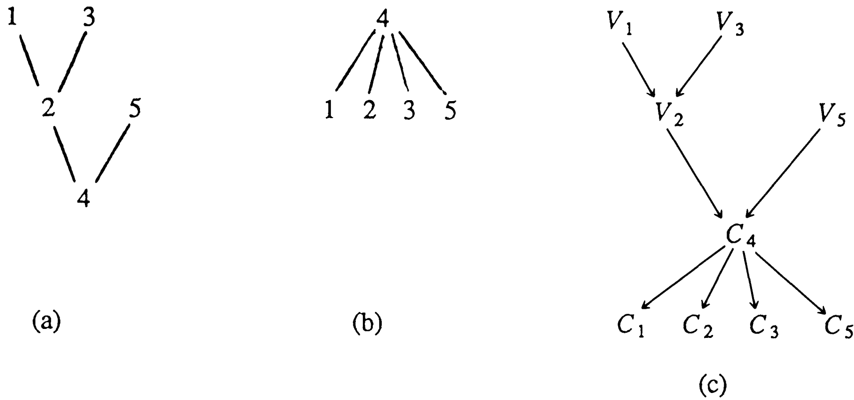


FIG. 3.3

THEOREM 1. *Let P be a set of participants, and c a set of associated communication costs. Then* $\min_{I \in \Psi} Cost(I) = 2 \cdot CMST(c, P)$.

Proof. Since the cost of an instance $I(k, T_1, T_1)$ for some minimum spanning tree T_1 is $2 \cdot CMST(c, P)$, then clearly $2 \cdot CMST(c, P) \geqq \min_{I \in \Psi} Cost(I)$. To obtain the inequality in the other direction, observe that, by Lemma 2, there exists a minimum communication cost instance, I^*, with one coordinator, say k. Based on I^*, construct an undirected graph, H, defined as follows. The nodes of H are the participants in P, and the edges of H are $\{(i, j) |$ there is a vote message from participant i to participant j in $I^*\}$. Since in I^* there is a path from the first event at every participant to C_k, H must be connected. Therefore its cost is at least $CMST(c, P)$, which in turn implies that the communication cost of vote messages in I^* is at least $CMST(c, P)$. Similarly, we can show that the cost of the commit messages of I^* is at least $CMST(c, P)$. Thus, $2 \cdot CMST(c, P) \leqq \min_{I \in \Psi} Cost(I)$. □

The result of Dwork and Skeen ([DS1, Thm. 1]) obtained for synchronous networks is extended to asynchronous networks by the following corollary of Theorem 1.

COROLLARY 1. *Assume that the number of participants in a transaction is p. If the communication cost of a message between each pair of participants is one, then it holds true that* $\min_{I \in \Psi} Cost(I) = 2(p-1)$. □

Hadzilacos obtained a similar extension of the result in the context of process failures and/or communication failures ([H2, Thm. 6]).

Now we shall present an algorithm that achieves minimum communication cost. In discussing commitment algorithms we assume, as in other works (e.g., [DS2]), that each participant knows the identity of all the participants, and we also assume that it knows the associated set of communication costs. The analysis of this section suggests a very simple minimum communication cost commitment algorithm, which we will call FIXED-COORDINATOR. It proceeds as follows. Each participant constructs some minimum spanning tree, T, of the cost graph, and selects a participant, k, designated as the coordinator. T and k are assumed to be identical at all the participants. This will be the case if the procedure which constructs T and selects k is identical at all the participants. The algorithm executes an instance with the vote messages corresponding to T' (which is the oriented tree with sink k, obtained by directing the edges of T towards k); the commit messages correspond to T'' (which is the rooted tree obtained from T by directing its edges away from k). Specifically, a participant $i \neq k$ waits until subtransaction completion and until the receipt of all vote messages represented by the arcs incoming into i in T'. Then it will send a "yes" vote message corresponding to the arc exiting i. Participant k, after completing its subtransaction, waits until receiving the votes from all its neighbors in T, and then commits. The propagation of the commit decision is similar, in the opposite direction.

Observe that the linear and central-site algorithms discussed in the introduction are special cases of the FIXED-COORDINATOR algorithm, in which the requirement that T is a *minimum* spanning tree is relaxed. In the central-site case, the tree consists of the coordinator connected by an edge to each other participant. In the linear case, the tree is simply a string, in which the coordinator is one of the leaves.

4. Characterization of commit instances with minimum communication cost. In this section we provide a complete characterization of all possible commit instances of minimum communication cost (Theorem 2). We determine that if a commit instance has a minimum communication cost, then each participant sends at most one vote message, and receives at most one commit message (a coordinator does not receive a commit message). Also, in a minimum communication cost instance there are either one or two coordinators. If there are two coordinators, then each participant sends exactly one vote message. If there is only one coordinator, then each participant except the coordinator sends one vote message and receives one commit message; the coordinator does not send a vote message. In either case, the messages of a minimum communication cost instance propagate "along edges" of minimum spanning trees of the cost graph. Specifically, there are two (not necessarily different) minimum spanning trees of the cost graph, such that the vote messages are only sent from a participant to its neighbor in one tree, and commit messages are only sent from a participant to its neighbor in the other. Furthermore, if the instance has two coordinators, then they must be neighbors in both trees; each one of them must send its vote message to the other, and no commit messages are sent between them.

A commit instance on spanning trees T_1 and T_2, coordinated at a participant k, was defined in § 3, and denoted $I(k, T_1, T_2)$. Similarly, we define next a commit instance on two spanning trees, coordinated at two participants. Assume that T_1 and T_2 are two spanning trees of the cost graph, such that participants m and n are neighbors in

both trees. A *commit instance on* T_1 *and* T_2 *coordinated at* m *and* n, denoted $I(m, n, T_1, T_2)$, is defined as follows. Denote by T_1' the graph obtained from T_1 by directing its edges to obtain an oriented tree with sink m, then omitting from it the arc $n \to m$. Denote by T_2' the graph obtained from T_2 by directing its edges to obtain a rooted tree with m as a root, then omitting from it the arc $m \to n$.

One obtains the instance $I(m, n, T_1, T_2)$ as a result of the following modifications:

(1) relabel the nodes from T_1'; node i is relabeled V_i;

(2) relabel the nodes from T_2'; node i is relabeled C_i;

(3) add the message arcs $V_m \to C_n$ and $V_n \to C_m$, and the order arcs $V_i \to C_i$ for each i.

Intuitively, $I(m, n, T_1, T_2)$ is a commit instance having vote messages that correspond to the arcs of T_1', plus the arcs $m \to n$ and $n \to m$, and commit messages that correspond to the arcs of T_2'. In other words, in $I(m, n, T_1, T_2)$ each participant, including the coordinators, sends exactly one vote message. The vote of n is received by m, and vice versa. Each participant, except the coordinators, receives exactly one commit message. Fig. 4.1 demonstrates the definition. It illustrates a minimum communication cost instance on T_1 and T_2 of Fig. 3.3, coordinated at participants 2 and 4. Note that $I(m, n, T_1, T_2)$ is the same graph as $I(n, m, T_1, T_2)$.

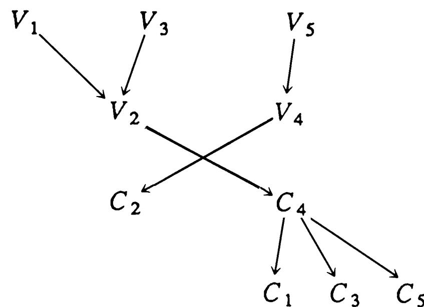


FIG. 4.1

THEOREM 2. *Let* P *be a set of participants, and let* c *be a set of associated communication costs. Any minimum communication cost instance* $I \in \Psi$ *must be of the form* $I(k, T_1, T_2)$ *or* $I(m, n, T_1, T_2)$, *for some participants* k, m, n, *and minimum spanning trees* T_1 *and* T_2 *of the cost graph.*

In order to prove Theorem 2 we shall show that: (a) any one-coordinator, minimum communication cost instance must be of the form $I(k, T_1, T_2)$; (b) any two-coordinator, minimum communication cost instance, must be of the form $I(i, j, T_1, T_2)$; and (c) a commit instance with three or more coordinators cannot have minimum communication cost. Fix the set of participants, P, and its associated set of communication costs, c, for the rest of this section.

LEMMA 4. *A one-coordinator minimum communication cost commit instance must be of the form* $I(k, T_1, T_2)$ *for some minimum spanning trees* T_1 *and* T_2.

Proof. Consider a minimum communication cost instance, I, with one coordinator, k. Following a line of reasoning similar to that used in the proof of Theorem 1, it can easily be seen that the total cost of vote messages is exactly $CMST(c, P)$, and the total cost of commit messages is exactly $CMST(c, P)$. Consider the undirected graph, H, having as nodes the participants and edges $\{(i, j)|$ there is a vote message from participant i to participant j in $I\}$. H must be connected, its cost is $CMST(c, P)$, and therefore it is a minimum spanning tree of the cost graph, T_1. Clearly, the vote messages

of I correspond to the oriented tree with sink k, obtained by directing the edges of T_1. Similarly we can show that the commit messages of I correspond to the rooted tree obtained by directing the edges of a minimum spanning tree, T_2. □

In Lemma 2 we presented a procedure to modify any minimum communication cost instance, I, with two or more coordinators, into another commit instance I' with one less coordinator, such that the communication costs of I and I' are identical. A similar procedure will be used here to characterize minimum communication cost instances with two or more coordinators.

LEMMA 5. *A two-coordinator minimum communication cost commit instance, I, must be of the form $I(i, j, T_1, T_2)$, for some participants i and j, and some minimum spanning trees, T_1 and T_2, both having the edge (i, j).*

Proof. Let C_i and C_j be the boundary C-events in I. By replacing one vote message by a commit message, as in Lemma 2, we obtain a one-coordinator instance of minimum communication cost. We then use the characteristics of such an instance given in Lemma 4 to show that I is indeed of the form $I(i, j, T_1, T_2)$.

Let V^0 denote a boundary V-event, as defined before the proof of Lemma 2. Since i and j are the only coordinators, V^0 precedes both C_i and C_j. Let C_i be the boundary C-event associated with V^0, and p_0 the participant at which V^0 occurs. Since there are only two boundary C-events, by definition of an "associated boundary C-event," C_{p_0} is in the tree of the C-subgraph which is rooted at C_j.

We first show that, in fact, p_0 is the coordinator j. By Lemma 3, observe that the path from V^0 to C_i is the message arc $V^0 \rightarrow C_i$. Consider the commit instance I' obtained from I by replacing $V^0 \rightarrow C_i$ by $C_{p_0} \rightarrow C_i$ (as in the proof of Lemma 2). I' has only one coordinator, j, and has the same cost as I, i.e., minimum communication cost. Therefore, by Lemma 4, the instance I' must be of the form $I(j, T_1, T_2)$ for some minimum spanning trees T_1, T_2. In particular, in the instance I', the coordinator j does not send a vote message, whereas in I, participant j must send a vote message, because there must be a path from the first event at j to C_i. However, the only vote message deleted in I to obtain I' is $V^0 \rightarrow C_i$; hence V^0 occurs at participant j, or, in other words, $p_0 \equiv j$.

Because of the new arc, $C_{p_0} \rightarrow C_i$, j and i are neighbors in T_2. We shall show that i is a neighbor of j in T_1 as well, or, in other words, that i sends its vote to j in I. Denote the single V-event at each participant r in I by V_r. We have already shown that if V^0 has C_i as an associated C-event, then V^0 is actually V_j. If there is no other boundary V-event in I, then all V-events must precede V_j, and therefore V_j should be a C- rather than a V-event. Consequently V_j cannot be the only boundary V-event in I. Denote by V^1 another boundary V-event, i.e., $V^1 \neq V^0$. The associated C-event of V^1 cannot be C_i, since otherwise, as above, we can show that $V^1 = V_j$, which in turn implies that $V^1 = V^0$. Thus, the associated C-event of V^1 must be C_j, and, as above, V^1 occurs at participant i; that is, V^1 is actually V_i, so participant i sends its vote to participant j.

Now, recall that I' is obtained from I simply by replacing $V_0 \rightarrow C_i$ by $C_j \rightarrow C_i$, and that I' has the form $I(j, T_1, T_2)$. Therefore, in the instance I, exactly one vote message is sent by each participant, and i is of the form $I(i, j, T_1, T_2)$ (by definition of such an instance). □

LEMMA 6. *There are no minimum communication cost commit instances with three or more coordinators.*

Proof. From the proof of Lemma 2, we know that from every minimum communication cost instance with two or more coordinators, it is possible to obtain a minimum communication cost instance with one less coordinator. Consequently, it is sufficient to show that there is no minimum communication cost instance with three coordinators.

Suppose that there exists a minimum communication cost instance, I, with three coordinators i, j, m. Let V^0 be a boundary V-event, and C_i an associated boundary C-event. Assume that V^0 occurs at participant p_0. By replacing $V^0 \to C_i$ with $C_{p_0} \to C_i$, we obtain, as in Lemma 2, an instance I' of minimum communication cost with two coordinators, j and m. By Lemma 5, instance I' must have the form $I(j, m, T_1, T_2)$, for some minimum spanning trees T_1 and T_2. Event V^0 occurs at exactly one participant, therefore it cannot occur at both coordinators. Assume without loss of generality that it does not occur at coordinator j, i.e., $p_0 \neq j$. Then the vote messages exiting V-events occurring at participant j are the same in $I(j, m, T_1, T_2)$ and I. However, by definition, the only such vote message in $I(j, m, T_1, T_2)$ is $V_j \to C_m$. Therefore, in I, the only V-event occurring at participant j does not precede boundary C-event C_i. Thus I does not satisfy the commit requirement, contradicting that fact that I is a commit instance. □

We have completed the proof of Theorem 2; as an immediate corollary we conclude the following.

COROLLARY 2. *If the communication cost of some commit instance with p participants is minimum, then it has a minimum number of intersite messages*, i.e., $2(p-1)$. □

Obviously, as demonstrated in the introduction, the converse of Corollary 2 is not true, i.e., a minimum number of intersite messages does not imply minimum communication cost.

The characterization in Theorem 2 helps us demonstrate that, in general, minimum communication cost cannot be achieved with limited knowledge of the participants' identity. For example, consider the network and the participants of Fig. 1.1. Suppose that each one of the participants 2, 3, and 4 knows only that 1 is a participant, but does not know of the existence of other participants. Suppose also that 1 knows that 4 is a participant, and that there are two other participants, but 4 does not know their identity. A possible commitment scenario is that participant 4 waits for the votes of all the other participants, and then propagates the commit decision. Another scenario is that 1 only waits until receiving the votes of the two anonymous participants, and then transmits their vote, along with its own, to 4; the latter then propagates the commit decision. In the two scenarios above, as well as in any other possible scenario, a message must be sent between 1 and 4. Since the edge 4–1 in the cost graph does not belong to any minimum spanning tree, minimum communication cost cannot be achieved.

5. Communication time of commit instances. In this section we first define the communication time of an instance, and then we establish the minimum communication time of a commit instance.

Generally, time comparison of instances in a totally asynchronous network is impossible, because each message can have an arbitrarily long delay. Therefore some restrictions on the network behavior must be imposed. The only restriction we impose here is that the delay of a message between every pair of participants is fixed for the duration of any commitment-algorithm execution. Thus, any message from i to j, sent by any algorithm, takes a fixed and finite amount of time to arrive, say t_{ij}.

The communication time of an instance on a set of participants, P, is defined with respect to a set $\tau = \{\tau_i | i \in P\}$ of *subtransaction completion times*, and with respect to a set $t = \{t_{ij} | i, j \in P\}$ of *intersite communication delays* (or *intersite delays* for short). For $i \neq j$, the intersite delay, t_{ij}, is a positive real number, and each τ_i is a nonnegative real number. The smallest τ_i is zero, and so is every t_{ii}. Each t_{ij} is the interval of time from the send of any message at i until its receive at j in any instance on P. We observe

that t_{ij} may be different than t_{ji}. (In the cost model, $c_{ij} = c_{ji}$, but our time results hold for the more general case.) The set t satisfies the triangle inequality, namely, for each i, j, k, $t_{ij} \leqq t_{ik} + t_{kj}$. (Our communication cost results hold even if the triangle inequality is not satisfied.) Let I be an instance on P. The sending of each message exiting an event, say e, happens at the execution time of e, defined as follows.

Let e be an event at participant i. The *execution time* of e, denoted $time(e)$, is the maximum of:

(1) The last (highest) receive of a message entering e, if there is any. The receive of a message exiting event e' at participant j occurs at $time(e') + t_{ji}$;

(2) The execution time of the event immediately preceding e at participant i, if there is any; and

(3) The subtransaction completion time, τ_i.

Observe that the execution times of events on any directed path in I are nondecreasing.

The *communication time* of the instance I, denoted $time(I)$, is defined as the maximum execution time of an event in I (i.e., the execution time of the last event). Intuitively, the communication time of I is the interval from the time that the first participant completes its subtransaction (taken to be zero), until the execution time of the last event, assuming the following. A participant i does not send its first message before time τ_i, and internal processing after subtransaction completion takes zero time at each participant. Clearly, for given sets of subtransaction completion times and intersite communication delays, the communication time may differ from one instance to another. For example, assume that all the subtransaction completion times are zero, and all the intersite delays are one. Then the communication time of the instance in Fig. 2.1(a) is two, whereas the communication time of the instance in Fig. 2.1(b) is one. (To contrast time with cost, if all pairwise costs are one, then the costs of the instances in Figs. 1.2(a) and 1.2(b) are four and six, respectively.) Given an instance I and sets τ and t, $time(I)$ can be computed in linear time by PERT techniques (see [E]).

PROPOSITION 1. *Let P be a set of participants, let τ be a set of subtransaction completion times, and let t be a set of intersite delays. Then the minimum communication time of an instance in Ψ (the set of commit instances for P) with respect to τ and t is*: $\max\{\tau_i + t_{ij} \mid i \text{ and } j \text{ are participants}\}$.

Proof. Consider some participant, i. In any instance I of Ψ, the execution time of the first event, say e, at participant i cannot be smaller than τ_i. Also, there must be a path from e to some event at every other participant. Since t satisfies the triangle inequality, $\max\{\tau_i + t_{ij} \mid j \text{ is a participant}\}$ is a lower bound on $time(I)$. i is an arbitrary participant, thus $\max\{\tau_i + t_{ij} \mid i \text{ and } j \text{ are participants}\}$ is a lower bound on $time(I)$. The lower bound is also an upper bound, since it is the communication time of the commit instance in which every participant, i, has one V_i event, and one C_i event, and there is a message from every V_i to every other C_j (e.g., the instance in Fig. 2.1(b)). (Observe that if some τ_k is bigger than $\tau_i + t_{ik}$, for each $i \neq k$, then the execution time of both V_k and C_k is τ_k.) □

If all the participants vote "yes," then clearly the decentralized algorithm executes a minimum communication time commit instance for any sets τ and t.

6. The TREE-COMMIT algorithm. TREE-COMMIT is a distributed, minimum communication cost algorithm, adapted from the PIF (Propagation of Information with Feedback) algorithm of [Se]. Each participant constructs the same minimum spanning tree, T, of the cost graph. In contrast to the fixed-coordinator algorithm, in TREE-COMMIT, a coordinator is not selected when the tree is constructed. The

procedure performed by each participant, i, in a committing execution is as follows. After subtransaction completion, it waits until receiving the votes from all its neighbors in T except one, say j, before voting; then it sends its vote to j. If i receives votes from all neighbors in T before it completes its subtransaction, then i commits and becomes the single coordinator. If i receives a vote message from j after having sent its vote to j, then it commits, becoming one of two coordinators (j is the other one). If i receives a commit message from j, then it sends commit messages to all its neighbors in T, except j. Therefore, the votes travel from the leaves of T inwards, where one or two coordinators are established. In the commit stage, the commit message is simply propagated along the tree edges, away from the coordinator(s).

A possible situation in the voting stage, i.e., before the coordinators are determined, is illustrated in Fig. 5.1(a). In the scenario illustrated, we suppose that participants 1, 2, and 5 have completed their subtransactions, and participants 3 and 4 have not. Participant 2 has not voted yet because it has not received the votes of two of its neighbors. Figs. 5.1(b) and 5.1(c) illustrate two possible instances executed by TREE-COMMIT at the completion of the voting stage situation described above. In the first case, 3 completed its subtransaction, and its vote had reached 2 before the vote of 4 did; furthermore, the votes of 2 and 4 crossed, so they both became coordinators. In the second case, 3 completed its subtransaction after having received the vote of all participants, so 3 became the sole coordinator.

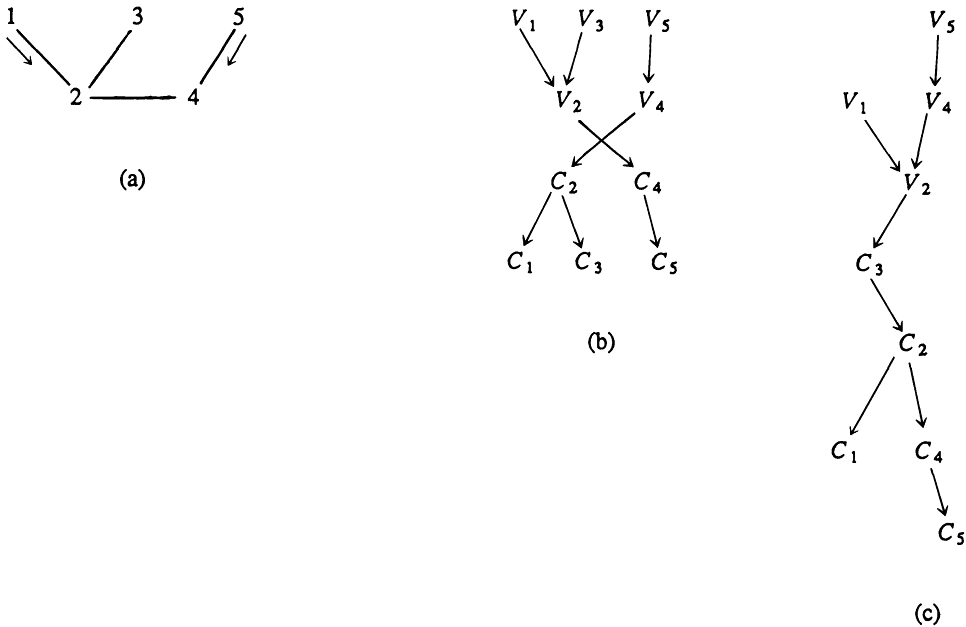


FIG. 5.1

We will denote by TREE-COMMIT (T) the algorithm which uses the tree T. The reader should be convinced that TREE-COMMIT (T) generates an instance of the form $I(k, T, T)$ or $I(m, n, T, T)$, for some coordinators m, n, k.

The transaction-abort case is handled by TREE-COMMIT as follows: Participant i sends "abort" messages when the first of the following two cases occurs: (i) i unsuccessfully completed its subtransaction, in which case i sends an "abort" (or a

"no" vote) message to each neighbor in T; (ii) i receives the first "abort" message from a participant, say k, in which case i sends an "abort" message to each neighbor except k.

The complete TREE-COMMIT algorithm for each participant, i, is given in Fig. 5.2. Note that TREE-COMMIT uses only "yes," "commit," and "abort" messages, therefore the message length can be restricted to two bits.

```
TREE-COMMIT (c, P)/* procedure executed by participant i, given a set of communication costs, c,
                                                      between participants in P*/
 1. Construct a minimum spanning tree, T, of the cost graph.
 2. Wait until subtransaction completion, or receipt of an "abort" message. /* vote-messages received are
                    saved, but they do not wake up this process; such messages are being considered in
                    steps 5, 6*/
 3.     if subtransaction completion then do;
 4.         if successful completion then do;
 5.           if received "yes" votes from all neighbors in T, then send
              "commit" messages to all neighbors, and quit. /* i is a single coordinator */
 6.           otherwise wait until receiving the first "abort" message, or until
              receiving a "yes" vote from all neighbors, except one.
 7.               if "abort" message from a neighbor, say k, then send "abort" messages to all
                  neighbors, except k, and quit.
 8.               if "yes" votes from all neighbors, except one, say j, then send a "yes" vote to j.
 9.         end.
10.         otherwise (unsuccessful completion) send "abort" messages to all neighbors, and quit.
11.     end.
12.     otherwise ("abort" message from a neighbor, say k) send "abort" messages to all neighbors,
        except k, and quit.
13. Wait until receiving a message.      /* the only way to get here is from step 8,
                                   and the message must have been received from j*/
14.     if "abort" message, then send "abort" messages to all neighbors, except j, and quit.
15.     otherwise ("commit" or "yes"-vote) send "commit" messages to all neighbors, except j,
        and quit. /* in the "yes"-vote case, i and j are coordinators */
```

FIG. 5.2

We have not formally defined an "abort" instance, but the next theorem indicates that the lower bound on its communication cost is lower than the commit instance lower bound.

THEOREM 3. *If some participant sends an "abort" message, then the communication cost of the instance executed by* TREE-COMMIT (T) *is at least the cost of T, and at most twice the cost of T.*

Proof. Clearly, commit messages are sent only if all the subtransactions completed successfully, and therefore, if some participant sends an "abort" message, then no commit messages are sent. Each message in the instance executed by TREE-COMMIT (T) is sent between two neighbors in T, therefore let us consider the edges of T. For each edge (i, j) there is either exactly one abort message from i to j, or exactly one "yes" vote message from i to j, but not both. The situation is similar from j to i. Hence the total cost of messages is at most twice the cost of T. Additionally, note that for each edge (i, j) of T there is an "abort" message from i to j, or from j to i. Hence the total communication cost of the instance is at least the cost of T. □

It is easy to see that the bounds of Theorem 3 are tight. If all the subtransactions complete unsuccessfully at exactly the same time, then two "abort" messages are sent along each edge of T, one in each direction, and the communication cost is twice the

cost of T. If, on the other hand, some participant completes its subtransaction unsuccessfully, and after this, the "abort" messages reach each participant before it completed its subtransaction, then one "abort" message is sent along each edge of T, and the communication cost is the cost of T.

The algorithm TREE-COMMIT assumes that each participant also knows the identities of all the other participants. It is possible that for some transactions this assumption does not hold true, and then TREE-COMMIT cannot be used. However, let us mention two frequent cases in which the assumption does hold true. First, it holds true in a fully replicated database, since then each participant knows that all the other local database managers are participants. Second, for simple update transactions that access only one data item (e.g., add $10,000 to an account which is replicated at four participants), the assumption holds true, since each participant usually stores a directory indicating where each data-item is replicated.

7. Analysis of TREE-COMMIT. In this section we first establish the communication time of TREE-COMMIT, and then we show that it is at most $p \cdot$ [minimum communication time of a commit instance] (§ 7.1). Then we define the communication complexity of an instance, we discuss the communication complexity of TREE-COMMIT, and we point out that it constitutes a polynomial-time approximation for an NP-complete problem (§ 7.2). Then we consider the other atomic commitment algorithms discussed in this paper, and compare their performance with TREE-COMMIT (§ 7.3).

7.1. The communication time. Given a set of participants and a spanning tree, T, namely a tree in which the nodes are the participants, a *commit instance on T* is a commit instance in which every message is from a processor to one of its neighbors in T. A commit instance on T will be called a T-instance for short. TREE-COMMIT obviously can propagate messages along any spanning tree (simply do not insist on "minimum" in step 1 of Fig. 5.2) and in this section we will speak in this broader context. We will show that for a spanning tree, T, TREE-COMMIT (T) executes the instance with the minimum communication time among all T-instances, given any sets of subtransaction completion times and intersite delays.

In this section, the edge (i, j) of an undirected tree often represents the two arcs $i \to j$ and $j \to i$; whether it does so or not will be clear from the context. Also, whenever we speak of the length of a path from one node to another in a tree, we assume that the length of each arc, $i \to j$, is the intersite communication delay t_{ij}. Given a set of intersite communication delays, t, a tree T in which the nodes are the participants, and a participant r, denote by d_r the longest simple path in T, from r to another node.

LEMMA 7. *Let P be a set of p participants, let T be a tree in which the nodes are the participants, let $\tau = \{\tau_1, \cdots, \tau_p\}$ be a set of subtransaction completion times, and let $t = \{t_{ij} | i, j \in P\}$ be a set of intersite communication delays. Then the communication time of a T-instance, I, with respect to τ and t, is at least* $\max_{i \in P} \{\tau_i + d_i\}$.

Proof. Let k be an arbitrary participant. There must be a path in I, from $V_{k,1}$ (or $C_{k,1}$, if k is a single coordinator in I) to every other $C_{i,1}$ (see § 2 for the definitions of $V_{i,j}$ and $C_{i,j}$). Since the messages of I are transmitted only between neighbors in T, the communication time of I cannot be lower than $time(V_{k,1}) + d_k$ (or $time(C_{k,1}) + d_k$). Additionally, $time(V_{k,1})$ and $time(C_{k,1})$ cannot be lower than τ_k. □

LEMMA 8. *Let P, T, τ, and t be as in Lemma 7. Then the communication time of (the instance generated by)* TREE-COMMIT (T) *for τ and t, is not higher than* $\max_{i \in P} \{\tau_i + d_i\}$.

Proof. The instance generated by TREE-COMMIT, denoted I, is of the form $I(k, T, T)$ or $I(m, n, T, T)$, and I in particular has one or two coordinators. If it has one coordinator, k, then the communication time of I is clearly $\{\tau_k + d_k\}$, and the theorem follows. Assume now that I has two coordinators, m and n. We shall show that in this case there also exists some participant, k, such that the communication time of I is $\{\tau_k + d_k\}$.

First, note that the communication time of I equals the execution time of some C-event, say C_q. The C-subgraph of I consists of two rooted trees, one at C_n, and the other at C_m. Suppose that C_q is the tree rooted at C_n. Then, clearly $time(C_q) = time(C_n) +$[distance in T from n to q]. Suppose now that the edge (m, n) is removed from T. Denote by T_m and T_n the resulting subtrees that contain m and n, respectively. Obviously, q is in T_n. A simple but important observation lies at the heart of this proof: Since, in TREE-COMMIT, participant n voted before receiving the vote message from m, then $time(C_n) = time(V_m) + t_{mn}$. In other words, $time(C_n)$ is the time of the receive of the message $V_m \rightarrow C_n$. This observation is crucial because, as we shall point out, there is a participant, k, in T_m, such that $time(V_m) = \tau_k +$[distance in T from k to m]. Since q and k are in T_n and T_m, respectively, and particularly since they are not in the same subtree, then $time(C_q) = \tau_k +$[distance in T from k to q].

Therefore, all that remains to prove is that there is a participant, k, in T_m, such that $time(V_m) = \tau_k +$[distance in T from k to m]. For this, consider $time(V_m)$. By definition, it is either equal to τ_m, in which case the proof is complete, or, there is a message, say $V_j \rightarrow V_m$, such that $time(V_j) + t_{jm} = time(V_m)$. In the latter case, consider V_j. It is either equal to τ_j, in which case, again, the proof is complete, or, there is a message, say $V_u \rightarrow V_j$, such that $time(V_u) + t_{uj} = time(V_j)$. Proceeding in this fashion, we must eventually encounter an event, V_k, such that $time(V_k) = \tau_k$. Then, clearly, $time(V_m) = \tau_k +$[distance in T from k to m]. □

From Lemmas 7 and 8 we immediately obtain the following theorem.

THEOREM 4. *Let P, T, τ, and t be as in Lemma 7. Then the communication time of (the instance generated by)* TREE-COMMIT (T) *for τ and t, is* $\max_{i \in P} \{\tau_i + d_i\}$. *Furthermore, this communication time is minimum among the communication times of all the T-instances for the same sets τ and t.*

Next we shall show that the communication time of TREE-COMMIT is at most p times bigger than the minimum communication time of an instance in Ψ.

THEOREM 5. *Let P, T, τ, and t be as in Lemma 7. Denote by $I_{\min}$ some instance in Ψ that has minimum communication time with respect to τ and t, and denote by I_{TC} the instance executed by* TREE-COMMIT (T) *for τ and t. Them $time(I_{TC})/time(I_{\min}) \leqq p$.*

Proof. Denote by i the participant for which $\tau_i + d_i = time(I_{TC})$. By Lemmas 7 and 8, we know that there is such a participant. By Proposition 1 we know that $time(I_{\min}) = \max_{k, j \in P} (\tau_k + t_{kj})$. Denote by $q \rightarrow r$ the longest arc in some longest simple path from i on T (i.e., some path of length d_i). Denote by A the ratio $\tau_i + d_i / \max_{k, j \in P} (\tau_k + t_{kj})$. Remember that there are at most $p-1$ arcs on a simple path from i, and consequently, $d_i \leqq (p-1) \cdot t_{qr}$. Therefore, if $\tau_i \geqq t_{qr}$, then the following holds true. $A \leqq p \cdot \tau_i / \max_{k, j \in P} (\tau_k + t_{kj})$; Obviously, $\max_{k, j \in P} (\tau_k + t_{kj}) \geqq \tau_i$; and consequently, $A \leqq p$. On the other hand, if $\tau_i < t_{qr}$, then the following holds true. $A \leqq p \cdot t_{qr} / \max_{k, j \in P} (\tau_k + t_{kj})$; obviously, $\max_{k, j \in P} (\tau_k + t_{kj}) \geqq t_{qr}$; and again, $A \leqq p$. □

Next we demonstrate that the bound on the ratio between the communication time of TREE-COMMIT and the minimum communication time is tight. Assume that all the subtransaction completion times are zero, all the intersite delays are one, and the tree, T_0, along which TREE-COMMIT propagates messages, is a string from 1 to p, namely, 1—2—3—, · · · , —p (note that p is the number of participants, as well as

the identification of the last processor). Then the time of TREE-COMMIT is $p-1$, whereas the minimum communication time is one.

7.2. The communication complexity. Let P be a set of participants, let c be a set of communication costs, let τ be a set of subtransaction completion times, and let t be a set of intersite communication delays. We define the *communication complexity* of an instance I on P, denoted $com(I)$, to be the product $cost(I) \cdot time(I)$. Note that it is possible for the minimum communication complexity commit instance not to be a minimum communication cost commit instance, nor a minimum communication time commit instance.

Since TREE-COMMIT on a minimum spanning tree has a minimum communication cost, we obtain the following corollary as an immediate consequence of Theorem 5.

COROLLARY 3. *Let P, τ, and t be as in Lemma 7. Let c be a set of costs associated with P, let I_{TC} be the instance generated by* TREE-COMMIT *on a minimum spanning tree of the cost graph, and denoted by $I_{\min}$ some instance in Ψ that has a minimum communication complexity with respect to c, τ, and t. Then $com(I_{TC})/com(I_{\min}) \leqq p$.*

It is easy to see that $O(p)$ is a tight bound on the ratio between the complexity of TREE-COMMIT and the optimal complexity. To realize this, simply consider again the tree T_0 (namely, 1—2—3—, $\cdots$, —p), and assume that all the communication costs and intersite delays are one, and all the subtransaction completion times are zero. Then the complexity of TREE-COMMIT (T_0) is $2(p-1)^2$, whereas the following instance has a complexity of $4(p-1)$. All the participants send their vote to 1, and 1 sends the commit message to all the other participants (see Fig. 2.1(a)). In other words, since the costs of the two instances are equal, but the time of TREE-COMMIT is $O(p)$ times higher, the complexity of TREE-COMMIT is $O(p)$ times higher than the minimum complexity.

Given P, τ, and t as in Lemma 7, and an integer, K, it is NP-complete to determine whether there exists an instance I in Ψ, for which $com(I) \leqq K$. In other words, the problem of finding the minimum complexity instance (MCI) is NP-complete.

For proof, observe first that the MCI problem is obviously in NP. A nondeterministic algorithm needs to guess an instance, and check whether its communication complexity is $\leqq K$. The MCI problem reduces to the MINIMUM RADIUS MINIMUM SPANNING TREE (MRMST)[4] problem defined as follows.

Input: A connected graph $H = (V, E)$, a vertex $r \in V$ (the root), a (weight) function w, which maps each edge of E to an integer and which satisfies the triangle inequality, and an integer k.

Question: Does H have a minimum spanning tree, T, such that the distance from r to any node is less than, or equal to, k.

Given an MRMST instance we construct an MCI instance as follows. Assume that m is the total weight of a minimum spanning tree of H. Let z be a node that is not in H. Define $P = V \cup \{z\}$. The set of communication costs, c, is defined as follows. For any pair of nodes i and j that are both in V, such that (i, j) is in E, $c_{ij} = c_{ji} = w(i, j)$. If i is z and j is r, or vice versa, then $c_{ij} = c_{ji} = 1$. Each one of the remaining costs, c_{ij}, is simply the length of the shortest path between i and j in the incomplete graph created thus far. The set of intersite delays, t, is defined as follows. For any pair of

[4] The MRMST problem was shown to be NP-complete by Itai ([I]). Note that the problem closely resembles the BOUNDED DIAMETER SPANNING TREE problem (ND4 in [GJ]).

nodes i and j that are both in V, $t_{ij} = c_{ij}$. Additionally, $t_{zr} = t_{rz} = k$. For each one of the remaining intersite delays, $t_{ij} = t_{ji}$ = *length of the shortest path between i and j, where the length of each edge* (r, q) *is* t_{rq}. The set of subtransaction completion times, τ, is defined as follows. If $i \neq r$ then $\tau_i = 0$; otherwise, $\tau_r = (2m)^4$. Finally, $K = 2(m+1)[(2m)^4 + k]$.

Using Theorem 2, it is not hard to see that the MCI instance has a solution if and only if the MRMST instance has a solution. Actually, the same transformation can be used to show that the following two problems are also NP-complete: Finding, among all the instances that have minimum communication cost, one that has a minimum communication time; and, finding among all the instances that have minimum communication time, one that has a minimum communication cost.

Now let us consider the computation time complexity of TREE-COMMIT, i.e., its time-complexity assuming that the communication time is zero (in contrast, remember that the communication-time is defined assuming that the computation- or internal processing time was zero). The computation time is dominated by the complexity of finding a minimum spanning tree, T, and consequently is $O(p^2)$ (see [E]). Corollary 3 means that TREE-COMMIT (T) is a polynomial-time, bounded error approximation algorithm for the MCI (minimum complexity instance) problem.

7.3. Comparison with other algorithms. First, consider the communication complexity and the communication cost of the decentralized algorithm (see § 5) that has minimum communication time. We shall demonstrate that the complexity (cost) can be $O(p^2)$ times the minimum communication complexity (cost). Assume that $T_0 = 1—2—3—, \cdots, —p$ is a minimum spanning tree of the cost graph, and the cost of each edge in T_0 is one. For every pair of participants, i and j, let the communication cost c_{ij} be the distance in T_0 between i and j. Then it is easy to see that the communication cost of the instance executed by the decentralized algorithm is $O(p^3)$, whereas the minimum communication cost is $2(p-1)$.

For demonstrating the complexity of the decentralized algorithm, let us continue this example, and suppose that for every pair of participants, i and j, $t_{ij} = c_{ij}$, and all subtransaction completion times are zero, except for the completion time of participant p, which is a very large number, say p^5. Then the communication complexity of the decentralized algorithm is $O(p^2)$ times the minimum communication complexity. An easy way to see this is to observe that TREE-COMMIT (T_0), in addition to having minimum communication cost, also has a minimum communication time.

Actually, it is easy to see that $O(p^2)$ is an upper bound on the ratio of the communication complexity (cost) of the decentralized algorithm to the minimum communication complexity (cost), assuming that the costs satisfy the triangle inequality. The reason for this is that the cost of each message is bounded by the cost of a minimum spanning tree of the cost graph, and the number of messages sent by the algorithm is $p(p-1)$.

Consider now the other two atomic commitment algorithms mentioned in the introduction, namely the *central-site* and the *linear* algorithms. As explained in § 3, they are both special cases of the FIXED-COORDINATOR algorithm. The communication cost of the FIXED-COORDINATOR algorithm, propagating messages along the edges of some tree, is equal to the communication cost of TREE-COMMIT, propagating messages along the edges of the same tree. Based on Theorem 4, the communication time of TREE-COMMIT (T) is never higher than the communication time of an arbitrary algorithm, particularly FIXED-COORDINATOR, in which messages are sent only between neighbors in T. Next, we shall establish the communica-

tion time of the FIXED-COORDINATOR algorithm. For this we need to define, for a given set of intersite communication delays t, a tree T in which the nodes are the participants, and two participants r and k, the *tree-distance* from k to r, denoted d_{kr}: it is the length of the path along the tree edges from k to r (remember, the length of $i \to j$ is t_{ij}).

THEOREM 6. *Let P be a set of participants, let r be a participant, and let T be a tree in which the nodes are the participants. Denote by I the instance generated by the* FIXED-COORDINATOR *algorithm that has the coordinator r, and propagates messages along the edges of the tree T. For any set of subtransaction completion times, τ, and any set of intersite delays, t, time(I) is* $\max_{i \in P} \{\tau_i + d_{ir}\} + d_r$.

Proof. By definition, $time(I)$ is $time(C_r) + d_r$. Additionally, we will show that $time(C_r) = \max_{i \in P} \{\tau_i + d_{ir}\}$. The proof for this is as follows. Obviously, $time(C_r)$ cannot be lower than $\max_{i \in P} \{\tau_i + d_{ir}\}$, since the vote of every participant must reach r along a path in the tree, and a participant, i, cannot send its vote message before time τ_i. However, $time(C_r)$ is not higher than $\max_{i \in P} \{\tau_i + d_{ir}\}$ for the following reason. By definition, $time(C_r)$ is either equal to τ_r, in which case the proof is complete, or, there is a message, say $V_j \to C_r$, such that $time(V_j) + t_{jr} = time(C_r)$. In the latter case, consider V_j. It is either equal to τ_j, in which case again the proof is complete, or, there is a message, say $V_u \to V_j$, such that $time(V_u) + t_{uj} = time(V_j)$. Proceeding in this fashion, we must eventually encounter an event, V_k, such that $time(V_k) = \tau_k$. Then, $time(C_r) = \tau_k + [\text{distance in } T \text{ from } k \text{ to } r]$. □

Therefore, the communication complexity (time) of FIXED-COORDINATOR on some tree, T, can be twice the communication complexity (time) of TREE-COMMIT (T). This happens if $\tau_r = 0$, and $\max_{i \in P} \{\tau_i + d_i\} = d_r$, and there is some participant k, such that $\{\tau_k + d_{kr}\} = d_r$. For example, consider the linear algorithm on some string, T, and assume that the set t is such that $t_{ij} = t_{ji} = 1$ for each edge (i, j) of T, and the subtransaction completion time of each participant is zero. Then, assuming that there are p participants, the communication time of TREE-COMMIT is $p-1$, whereas the communication time of the linear algorithm is $2(p-1)$. The communication time of the central site algorithm cannot be exactly twice the communication time of TREE-COMMIT (since the requirement $\{\tau_k + d_{kr}\} = d_r$ necessitates that the coordinator is a leaf), but it can be arbitrarily close to twice the communication time of TREE-COMMIT. This happens, for example, if there is some participant, i, such that $t_{ir} = t_{ri} = 1$ for each other participant, j, $t_{jr} = t_{rj} = \varepsilon$, and such that all subtransaction completion times are zero. Then the communication time of the central-site algorithm is two, whereas in TREE-COMMIT, r sends its vote to i at time ε, and it receives i's vote at time 1; i and r become coordinators, and the communication time of TREE-COMMIT is $1 + \varepsilon$.

Finally, we shall point out that the communication time of the FIXED-COORDINATOR algorithm cannot be more than twice the communication time of TREE-COMMIT. To realize this, observe that $\max_{i \in P} \{\tau_i + d_{ir}\} \leqq \max_{i \in P} \{\tau_i + d_i\}$, and also $d_r \leqq \max_{i \in P} \{\tau_i + d_i\}$.

8. Discussion.

8.1. Conclusion. In this paper we discussed the communication cost, the communication time, the communication complexity, and the computation time complexity of atomic commitment algorithms. We established that the lower bound on the communication cost for solving the atomic commitment problem is twice the weight of a minimum spanning tree of the cost graph. Given a set of intersite delays, t, and a set of subtransaction completion times, τ, the lower bound on the communication time is

$\max\{\tau_i + t_{ij} | i$ and j are participants$\}$. TREE-COMMIT, a new atomic commitment algorithm introduced in this paper, achieves, in the absence of failures, minimum communication cost. The decentralized algorithm achieves, in the absence of failures, minimum communication time. We also characterized the minimum communication cost commit instances, and showed that each such instance must propagate messages along two (not necessarily different) minimum spanning trees of the cost graph, and that it must have one or two coordinators.

Then we analyzed TREE-COMMIT, and we compared it with existing variants of the two-phase commit paradigm, namely, the decentralized, the linear, and the central-site algorithms. The communication time of TREE-COMMIT is at most p times the minimum communication time, where p is the number of participants. Consequently, its communication complexity, the product of its communication cost and its communication time, is also at most p times the minimum communication complexity (an NP-complete concept). Furthermore, the computation time of TREE-COMMIT is polynomial in the size of the input. The minimum communication time algorithm, namely the decentralized one, is also an approximation of the minimum communication complexity, but its error can be of order p^2.

When compared with the linear and the central-site algorithms, which are special cases of the FIXED-COORDINATOR algorithm, TREE-COMMIT is not only better in the worst case, but it is better in any case, in the following sense. Its communication cost can be made equal to the communication cost of FIXED-COORDINATOR, by parameterizing TREE-COMMIT to propagate its messages along the same spanning tree as FIXED-COORDINATOR. Then, the communication time of TREE-COMMIT cannot be worse than the communication time of FIXED-COORDINATOR for any sets of the intersite delays and subtransaction completion times. Furthermore, for some sets of the intersite delays and subtransaction completion times, the communication time of TREE-COMMIT is half the communication time of FIXED-COORDINATOR.

8.2. Future work. Society as a whole becomes increasingly dependent on communication and information dissemination. Also, in business, government, and the military, transaction processing gains ground, often at the expense of other types of processing. Therefore, we feel that the issues of gossiping, atomic commitment, and similar problems, will become increasingly important, and we intend to continue the study initiated in this paper.

First, it is important to extend our results to handle failures. Some related questions are the following: What is the necessary communication cost and communication time of atomic commitment algorithms under different failure assumptions? Are they sufficient? How do existing algorithms (e.g., three phase commit) approximate the optimal communication complexity?

Second, it is interesting to determine the bounds on performance of algorithms having different levels of knowledge. For example, it is easy to extend our model to define abort instances. Intuitively, it is clear that if there is exactly one "no"-voter, then a communication cost of one minimum spanning tree is necessary for abort. However, it is also intuitively clear that unless the participants know that there is one "no"-voter, in which case, executing an atomic commitment algorithm is obviously superfluous, this necessary cost is not sufficient. We do not know how to prove this yet. Alternatively, consider the example at the end of § 4. How do we determine the performance lower bounds, given limited knowledge of participants' identities? We feel that the interesting approach taken by Hadzilacos in [H2] provides a solid foundation for solving these problems.

Finally, on a more technical note, we will point out that the results in this paper suggest the solution to another variant of the "gossiping" problem (see § 1.3) that is unsolved in the literature (see [HHL]). In this variant, the pairs of individuals communicate by telephone calls or two-way sessions, as opposed to messages, or one-way sessions; in one call the two individuals *exchange* the pieces of the gossip known to them, rather than one transmitting to the other. The problem is, again, to find a solution that has minimum total cost (now c_{ij} is the cost of a telephone call between i and j). Notice that modelling an instance by a directed acyclic graph may be inappropriate for communication by two-way sessions. When the cost of each call is one (or when counting the number of calls), then the following algorithm achieves the minimum (necessary) cost, i.e., $2p-4$ calls (see [BK]).

MINIMUM-NUMBER-OF-CALLS:
1. Partition the individuals into four groups, A, B, C, and D, and appoint a leader in each group. Denote the leaders by a, b, c, and d, respectively.
2. Each leader calls every member in its group, collecting the information-piece from each one ($p-4$ calls).
3. a and b exchange information in one phone call, and so do c and d (2 calls).
4. a and c exchange information in one phone call, and so do b and d. At this point a, b, c, and d know the whole gossip.
5. Each leader calls every member in its group, telling each one the whole gossip ($p-4$ calls).

When the telephone-call costs differ from one pair to another, we conjecture that determining the minimum cost of a solution is NP-complete, and the reason will become clear in the discussion below. (Remember, for one-way communication the minimum cost can be determined in linear time.) Furthermore, we conjecture that there are two possible structures for the pattern of telephone calls that achieves minimum cost. The first is a pattern that resembles the one that has a minimum number of calls. Specifically, the individuals are partitioned into four groups, as in the algorithm MINIMUM-NUMBER-OF-CALLS. However, the information-piece (corresponding to the vote) of each member of a group, rather than being communicated directly to the group leader, flows to it along the edges of a minimum spanning tree. Then the leaders exchange information as in steps 3 and 4 of MINIMUM-NUMBER-OF-CALLS, and afterwards, the whole gossip flows back to all the individuals along the edges of the same four minimum spanning trees. We conjecture that finding a partition as above, for which the total cost of the telephone calls is minimum, is NP-complete.

The second possible structure for the pattern of minimum cost is one that corresponds to the edges of a spanning tree, T, of the cost graph. The gossip pieces flow inwards, towards the coordinators, which are the neighbors on both sides of the costliest edge in T. After a coordinator receives the pieces of the gossip from all its neighbors in the tree, except from the other coordinator, it calls the other coordinator, and they exchange information. At this point, the coordinators (and only them) know the whole gossip, the total cost is equal to the cost of T, and the number of calls is $p-1$. Subsequently, the gossip flows to the rest of the participants along the edges of T, requiring $p-2$ additional calls. The cost of these additional calls is: (the cost of T)-(the cost of the edge between the coordinators).

Next we will make a few remarks about the second pattern of minimum cost mentioned above. First, note that the total cost of the pattern above is:

(8.2.1) (twice the cost of T)-(the cost of the edge between the coordinators),

and this is the reason for choosing the coordinators as the endpoints of the costliest edge in T. Second, observe intuitively that there are cases in which the pattern above indeed achieves minimum communication cost. For example, it does so if there is a unique minimum spanning tree of the cost graph, and when any telephone call between participants that are not neighbors in the tree is prohibitively expensive. Third, the pattern above achieves minimum cost with $p-3$ calls, and this is higher than the minimum number of calls, that is $p-4$. This corroborates the argument made in § 3, that the communication pattern having a minimum number of messages (or calls) does not necessarily have a minimum cost. Fourth, we conjecture that, in general, finding the spanning tree for which formula (8.2.1) is minimum, is NP-complete.

In conclusion, we feel that much remains to be done in order to prove the conjectures above, and to then incorporate communication time considerations into the solutions. We believe that TREE-COMMIT will provide a handle on an approach for this incorporation.

Acknowledgments. We thank Alon Itai for helpful discussions and insightful remarks. We also thank the referees for detailed and thorough reports, which helped us improve this paper very significantly.

REFERENCES

[BHG] P. BERNSTEIN, V. HADZILACOS, AND N. GOODMAN, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[BK] B. BAKER AND R. SHOSTAK, *Gossips and telephones*, Discrete Math., 2 (1972), pp. 191–193.

[C] N. COT, *Extensions of the telphone problem*, in Proc. 7th SE Conference on Combinatorics, Graph Theory and Computing, Utilitas Mathematica, Winnipeg, Manitoba, Canada 1976, pp. 239–256.

[CP] S. CERI AND G. PELAGATTI, *Distributed Database Principles and Systems*, McGraw-Hill, New York, 1984.

[DS1] C. DWORK AND D. SKEEN, *The inherent cost of nonblocking commitment*, in Proc. 2nd ACM Symposium on Principles of Distributed Computing, 1983, pp. 1–11.

[DS2] ———, *Patterns of communication in consensus protocols*, in Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 143–153.

[E] S. EVEN, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[F] M. FISCHER, *The consensus problem in unreliable distributed systems* (*a brief survey*), Tech. Report YALEU/CDS-/RR-273, Yale University, New Haven, CT, June 1983.

[G] J. N. GRAY, *Notes on database operating systems*, Operating Systems: An Advanced Course, Springer-Verlag, Berlin, New York, 1979.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman, San Francisco, CA, 1979.

[H1] V. HADZILACOS, *On the relationship between the atomic commitment and consensus problems*, in Proc. Workshop on Fault-Tolerant Distributed Computing, Springer-Verlag, Berlin, New York, 1986.

[H2] ———, *A knowledge theoretic analysis of atomic commitment protocols*, in Proc. 6th ACM Symposium on Principles of Database Systems, San Diego, CA, 1987, pp. 129–134.

[HHL] S. HADETNIEMI, S. HADETNIEMI, AND A. LIESTMAN, *A survey of gossiping and broadcasting in communication networks*, Networks, Vol. 18, John Wiley, New York, 1988, pp. 319–349.

[I] A. ITAI, unpublished result, 1986.

[L] L. LAMPORT, *Time, clocks, and the ordering of events in a distributed system*, Comm. ACM, 21 (1978), pp. 558–565.

[La] B. W. LAMPSON, *Atomic Transactions*, in Distributed Systems—Architecture and Implementation, B. W. Lampson, M. Paul, and H. J. Siegert, eds., Springer-Verlag, Berlin, New York, 1981, pp. 247–265.

[LHJLSW] B. LISKOV, M. HERLIHY, P. JOHNSON, G. LEAVENS, R. SCHEIFLER, AND W. WEIHL, *Preliminary Argus Reference Manual*, Programming Methodology Group Memo 39, MIT Laboratory for Computer Science, Cambridge, MA, 1983.

[MLO] C. MOHAN, B. LINDSAY, AND R. OBERMACK, *Transaction Management in the R* Distributed Database Management System*, Transactions on Database Systems, 11 (1986), pp. 378–396.

[R] K. V. S. RAMARAO, *On the complexity of commit protocols*, in Proc. 4th ACM Symposium on Principles of Distributed Computing, Portland, OR, 1985, pp. 235–244.

[Se] A. SEGALL, *Distributed network protocols*, IEEE Trans. Inform. Theory, Vol. IT-29, 1983, pp. 23–35.

[Sk] D. SKEEN, *Nonblocking commit protocols*, in Proc. ACM-SIGMOD International Conference on Management of Data, 1981, pp. 133–142.

[Sp] A. SPECTOR, *Modular architectures for distributed and database systems*, in Proc. 8th ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, 1989, pp. 217–224.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 451-459, June 1991

© 1991 Society for Industrial and Applied Mathematics
003

SOME LOWER AND UPPER COMPLEXITY BOUNDS FOR GENERALIZED FOURIER TRANSFORMS AND THEIR INVERSES*

ULRICH BAUM† AND MICHAEL CLAUSEN†

Abstract. For $2 \leqq c \leqq \infty$, the c-linear complexity $L_c(A)$ of a complex matrix A is defined as the minimal number of additions, subtractions, and multiplications by complex constants of absolute value $\leqq c$, sufficient to evaluate A at a generic input vector. It is shown that if A is a Fourier transform on the finite group G, then $|L_\infty(A) - L_\infty(A^{-1})| \leqq |G|$. $L_c(G) := \min \{L_c(A) | A \text{ a Fourier transform for } G\}$ is called the c-linear complexity of the finite group G. It is proved that $L_2(G) > \frac{1}{4}|G| \log |G|$ for any finite group G, and two infinite classes of non-abelian groups G with $L_2(G) \leqq 0.6|G| \log |G|$ and $L_2(G) \leqq 0.8|G| \log |G|$, respectively, are presented. Thus there are non-abelian groups with even faster Fourier transforms than elementary abelian 2-groups (for which $L_2(G) \leqq |G| \log |G|$)!

Key words. fast Fourier transforms, fast inverse Fourier transforms, group algebras, Frobenius groups, extra-special 2-groups, linear complexity

AMS(MOS) subject classifications. 68Q40, 20C15

1. Introduction. The design and analysis of efficient algorithms for Fourier transforms on finite groups has been the subject of several recent investigations [1]-[4], [6]-[8], [11], [12], [14], [17], [19], [21], [22]. The present paper continues the studies in [6]. Although we assume familiarity with [6] and its notation, we briefly recall the mathematical background [10]-[13].

By Wedderburn's theorem, the group algebra $\mathbb{C}G$ of a finite group G is isomorphic to an algebra of block diagonal matrices: $\mathbb{C}G \simeq \bigoplus_{i=1}^{h} \mathbb{C}^{d_i \times d_i}$, where the blocks correspond to the equivalence classes of irreducible representations of $\mathbb{C}G$. Every algebra isomorphism $A: \mathbb{C}G \to \bigoplus_{i=1}^{h} \mathbb{C}^{d_i \times d_i}$ is called a (generalized) *Fourier transform* for $\mathbb{C}G$. With respect to natural bases, A can be viewed as a $|G|$-square complex matrix. (For example, if $G = C_n$ is the cyclic group of order n, then $A = (\omega^{ab})_{0 \leqq a,b < n}$ with $\omega = \exp(2\pi i/n)$.)

Thus a fast Fourier transform amounts to an efficient algorithm for evaluating a fixed matrix A at a generic input vector x. The linear computational model is a suitable algebraic framework for the analysis of this linear problem as follows. For $2 \leqq c \leqq \infty$, the *c-linear complexity* $L_c(A)$ of a matrix $A \in \mathbb{C}^{r \times t}$ is defined as the minimal number of linear operations (complex additions, subtractions, and scalar multiplications) sufficient to compute Ax from a generic input vector $x \in \mathbb{C}^t$, where scalar multiplications are restricted to complex constants of absolute value $\leqq c$. (Note that L_∞ corresponds to L_s in earlier papers. The condition $c \geqq 2$ is needed for consistency as $z + z = 2z$ is always computable in one step.)

Since a non-abelian group G has infinitely many Fourier transforms, we define the c-linear complexity of G by $L_c(G) := \min L_c(A)$, where the minimum is taken over all possible Fourier transforms A for $\mathbb{C}G$.

Obviously, $L_c(G) \geqq L_d(G)$ for $2 \leqq c \leqq d \leqq \infty$. Thus when looking for upper bounds, the L_2-model should be preferred. A closer look at [2] and [6] (which are based on the classical papers [5], [9], [16], [20], [23], [24]) shows that

$$L_2(G) \leqq 8|G| \log |G|$$

for all finite metabelian groups G, i.e., groups G whose commutator subgroup G' is abelian. (Throughout this paper, $\log = \log_2$.) If G is an abelian 2-group, then the

* Received by the editors February 28, 1990; accepted for publication (in revised form) September 6, 1990.
† Institut für Informatik V, Universität Bonn, 5300 Bonn, Federal Republic of Germany.

classical FFT-algorithms show that

$$L_2(G) \leqq \tfrac{3}{2}|G| \log |G|.$$

For the class of elementary abelian 2-groups, i.e., $G \simeq C_2 \times \cdots \times C_2$, the fast Hadamard–Walsh transforms prove the even better bound

$$L_2(G) \leqq |G| \log |G|.$$

What about lower bounds? As $L_c(G) \geqq L_\infty(G)$ for any $c \geqq 2$, we would be particularly interested in general nonlinear lower bounds for $L_\infty(G)$. Unfortunately, no such bounds are known. Yet for $c < \infty$, there is a result by Morgenstern [18] stating that $L_c(A) \geqq \log_c |\det A|$ for any invertible matrix A. The logarithm to the base c reflects a trade-off between the lower bound and the power of the computational model. In this paper, we will restrict ourselves to the L_2-model. This not only gives high lower bounds, but also allows us to compare lower and upper bounds.

In order to get a lower bound for the 2-linear complexity of a finite nonabelian group G, we are faced with the problem of estimating $L_2(A)$ for infinitely many Fourier transforms A on G. Nevertheless, combining Morgenstern's theorem and the Schur relations we can prove in § 3 that

$$L_2(G) \geqq \frac{1}{4}\left(1 + \frac{1}{|G'|}\right)|G| \log |G| > \frac{1}{4}|G| \log |G|$$

for any finite group G.

How tight is this general lower bound? Up to now, the elementary abelian 2-groups satisfying $L_2(G) \leqq |G| \log |G|$ seemed to have the fastest Fourier transforms. But, in fact, this is not true: In §§ 4 and 5 we present two infinite classes of non-abelian groups G with $L_2(G) < 0.6|G| \log |G|$ and $L_2(G) < 0.8|G| \log |G|$, respectively.

For many applications, e.g., fast convolution and digital filtering, the inverse of a Fourier transform is equally important. We show in § 2 that the linear complexities of any Fourier transform A and its inverse differ at most by the order of the group.

2. Schur relations and linear complexity. In this section we will use the classical Schur relations to prove a close connection between a Fourier transform and its inverse, which leads to some new lower and upper complexity bounds. To begin with, we recall the Schur relations.

SCHUR RELATIONS [13, V, Satz 5.7]. *Let $D_1, \cdots, D_h$ be a full set of inequivalent irreducible matrix representations of $\mathbb{C}G$ of degrees $d_1, \cdots, d_h$, respectively. Then for all $1 \leqq a, b \leqq h$ and $1 \leqq i, j \leqq d_a$ and $1 \leqq k, l \leqq d_b$, the following holds*:

$$\sum_{g \in G} D_a(g)_{ij} \cdot D_b(g^{-1})_{kl} = \delta_{ab}\delta_{il}\delta_{jk}\frac{|G|}{d_a}.$$

Note that the right-hand sides of the Schur relations only depend on the equivalence classes of the irreducible representations of $\mathbb{C}G$.

If A is a Fourier transform matrix for the finite group G, then the inverse of A is very similar to the transpose $A^\top$. More precisely, we have the following theorem.

THEOREM 1. *If A is a Fourier transform matrix for the finite group G, $\mathbb{C}G \simeq \bigoplus_{i=1}^h \mathbb{C}^{d_i \times d_i}$, then there exist permutation matrices P and Q such that*

$$A \cdot P \cdot A^\top \cdot Q = \bigoplus_{i=1}^{h} \frac{|G|}{d_i} \cdot E_{d_i^2},$$

where E_d denotes the d-square unit matrix.

Proof. Let G be a finite group of order n, and $D_1, \cdots, D_h$ a full set of inequivalent irreducible representations of $\mathbb{C}G$, $d_i :=$ degree (D_i). If $A \in \mathbb{C}^{n \times n}$ is a Fourier transform

matrix of $\mathbb{C}G$ with respect to $D_1, \cdots, D_h$, then the columns of A are parametrized by the elements of G whereas the rows of A correspond to

$$\bigcup_{1 \leqq a \leqq h} \{(a, i, j) | 1 \leqq i, j \leqq d_a\},$$

i.e., (a, i, j) describes the position (i, j) in D_a. Now let $B := P \cdot A^\top \cdot Q \in \mathbb{C}^{n \times n}$ be the matrix obtained from A by first transposing A and then performing in $A^\top$ the permutation P of the rows corresponding to the inversion $(G \ni g \mapsto g^{-1})$ and the permutation Q of the columns corresponding to $(a, i, j) \mapsto (a, j, i)$. According to the Schur relations, $A \cdot B$ is a diagonal matrix with d_a^2 occurrences of $|G|/d_a$ for $1 \leqq a \leqq h$. □

The following example will illustrate (the proof of) Theorem 1.

Example. The symmetric group S_3 has (up to equivalence) three irreducible $\mathbb{C}$-representations: the trivial representation

$$\iota : (S_3 \ni \pi \mapsto 1),$$

the alternating representation

$$\varepsilon : (S_3 \ni \pi \mapsto \operatorname{sgn}(\pi)),$$

and a two-dimensional representation Δ realizing S_3 as the symmetry group of a regular triangle. If we take its center of gravity as the origin in 2-space and denote its vertices by e_1, e_2, e_3, then $\{e_1, e_2\}$ is a basis and $e_3 = -e_1 - e_2$. The natural S_3-action $\pi e_i := e_{\pi i}$ yields the following realization of Δ:

$$\Delta(1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \Delta(123) = \begin{pmatrix} 0 & -1 \\ 1 & -1 \end{pmatrix}, \quad \Delta(132) = \begin{pmatrix} -1 & 1 \\ -1 & 0 \end{pmatrix},$$

$$\Delta(12) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \Delta(23) = \begin{pmatrix} 1 & -1 \\ 0 & -1 \end{pmatrix}, \quad \Delta(13) = \begin{pmatrix} -1 & 0 \\ -1 & 1 \end{pmatrix}.$$

Thus

$A =$

(1)	(123)	(132)	(12)	(23)	(13)	
1	1	1	1	1	1	ι
1	1	1	−1	−1	−1	ϵ
1	0	−1	0	1	−1	Δ_{11}
0	−1	1	1	−1	0	Δ_{12}
0	1	−1	1	0	−1	Δ_{21}
1	−1	0	0	−1	1	Δ_{22}

is a Fourier transform on S_3. The corresponding matrix B reads as follows:

$B =$

ι	ϵ	Δ_{11}	Δ_{21}	Δ_{12}	Δ_{22}	
1	1	1	0	0	1	(1)
1	1	−1	−1	1	0	(132)
1	1	0	1	−1	−1	(123)
1	−1	0	1	1	0	(12)
1	−1	1	0	−1	−1	(23)
1	−1	−1	−1	0	1	(13)

and $A \cdot B$ equals the diagonal matrix diag $(6, 6, 3, 3, 3, 3)$.

We will apply Theorem 1 to obtain bounds on the linear complexity of inverse Fourier transforms. To this end, we need a result due to Kaminski, Kirkpatrick, and Bshouty [15], which in our terminology reads as follows.

THEOREM 2. *Let* $A \in \mathbb{C}^{p \times q}$ *and* $c \geqq 2$. *Then*

$$L_c(A^\top) = L_c(A) - q + p - z(A) + z(A^\top),$$

where $z(A)$ denotes the number of all-zero rows of A. In particular, $L_c(A)=L_c(A^\top)$ for any invertible matrix A.

Combining Theorems 1 and 2, we get Theorem 3.

THEOREM 3. *Let A be a Fourier transform matrix for the finite group G. Then*
(1) $L_c(A^{-1})\leqq L_c(A)+|G|$ *for any* $c\geqq 2$.
(2) $|L_\infty(A)-L_\infty(A^{-1})|\leqq |G|$.

Proof. By Theorem 1,

$$A^{-1}=P\cdot A^\top\cdot Q\cdot\left(\bigoplus_{i=1}^{h}\frac{d_i}{|G|}E_{d_i^2}\right).$$

As A is invertible, we have $L_c(A^\top)=L_c(A)$ according to Theorem 2. Since $L_c(P)=0$ for permutation matrices P and $L_c(M_1\cdot M_2)\leqq L_c(M_1)+L_c(M_2)$ for arbitrary n-square matrices M_1 and M_2, our claims follow easily. □

3. Lower bounds in the L_2-model. In this section we will prove a general lower bound for the 2-linear complexity of any Fourier transform on a finite group G.

THEOREM 4. *If G is a finite group, $\mathbb{C}G\simeq\bigoplus_{i=1}^{h}\mathbb{C}^{d_i\times d_i}$, then*

$$L_2(G)\geqq\log\frac{|G|^{|G|/2}}{\prod_{i=1}^{h}d_i^{d_i^2/2}}.$$

Proof. Let A be a Fourier transform matrix for G. Then, according to Morgenstern's theorem [18], $L_2(A)\geqq\log|\det A|$. Define $B=P\cdot A^\top\cdot Q$ as in the proof of Theorem 1. By Theorem 1, $A\cdot B$ is a diagonal matrix with d_a^2 occurrences of $|G|/d_a$ for $1\leqq a\leqq h$. As $\det B=\pm\det A^\top=\pm\det A$, we get

$$|\det A|^2=|\det A\cdot B|=\prod_{i=1}^{h}\left(\frac{|G|}{d_i}\right)^{d_i^2}$$
$$=\frac{|G|^{d_1^2+\cdots+d_h^2}}{\prod_i d_i^{d_i^2}}=\frac{|G|^{|G|}}{\prod_i d_i^{d_i^2}}.$$

By Morgenstern's theorem, our claim follows. □

This result has several interesting consequences.

COROLLARY 1. *For any finite group G,*

$$L_2(G)\geqq\frac{1}{4}\left(1+\frac{1}{|G'|}\right)|G|\log|G|>\frac{1}{4}|G|\log|G|.$$

Proof. $\mathbb{C}G\simeq\bigoplus_{i=1}^{h}\mathbb{C}^{d_i\times d_i}$ implies $|G|=\sum_{i=1}^{h}d_i^2$. As the number of one-dimensional irreducible representations of G equals $[G:G']$ (see, e.g., [13, V, 6.5]), we get

$$\sum_{d_i>1}d_i^2=|G|-[G:G'].$$

Hence, by Theorem 4,

$$L_2(G)\geqq\frac{|G|}{2}\log|G|-\sum_{d_i>1}\frac{d_i^2}{4}\log d_i^2$$
$$\geqq\frac{|G|}{2}\log|G|-\sum_{d_i>1}\frac{d_i^2}{4}\log|G|$$
$$\geqq\frac{|G|}{2}\log|G|-\frac{1}{4}\left(1-\frac{1}{|G'|}\right)|G|\log|G|$$
$$=\frac{1}{4}\left(1+\frac{1}{|G'|}\right)|G|\log|G|.$$

This proves Corollary 1. □

COROLLARY 2. *If G is abelian, then*

$$L_2(G) \geqq \tfrac{1}{2}|G| \log |G|.$$

Proof. Use Theorem 4 and the fact that all $d_i = 1$ for an abelian group G. □

As a special case of Corollary 1 we mention Corollary 3.

COROLLARY 3. *If $|G'| = 2$ then*

$$L_2(G) \geqq \tfrac{3}{8}|G| \log |G|.$$

In the following two sections, we will present two classes of finite groups with very fast Fourier transforms. In order to prove the upper bounds, we will use some classical representation theory. For a short review of the tools and notation needed, the reader is referred to [6, pp. 588–590].

4. FFT for a class of Frobenius groups. We are going to consider a special class of Frobenius groups G_n constructed as follows. For $n \geqq 2$, let F_n be the additive group of the finite field $GF(2^n)$ and let H_n denote the multiplicative group of that field. It is well known that F_n is an elementary abelian group of order 2^n and H_n is cyclic of order $2^n - 1$. H_n acts faithfully on F_n by automorphisms via $H_n \ni h \mapsto (F_n \ni f \mapsto hf)$. As $hf \neq f$ for every $h \in H_n \backslash \{1\}$ and $f \in F_n \backslash \{0\}$, H_n can be viewed as a fixed-point-free automorphism group of F_n. Hence the semidirect product $G_n := F_n H_n$ is a Frobenius group of order $2^n(2^n - 1)$ (see, e.g., [13, V, § 8]). The ordinary representation theory of Frobenius groups is well understood (see, e.g., [13, V, Satz 16.13]). In our case, G_n has (up to equivalence) exactly the following irreducible representations:

(1) $2^n - 1$ one-dimensional representations χ_i $(1 \leqq i < 2^n)$ obtained by composing each linear character $\eta_i \in X(H_n)$ of the cyclic group H_n with the natural projection $G_n \to H_n \simeq G_n / F_n$, i.e.,

$$\chi_i(fh) := \eta_i(h)$$

for all $f \in F_n$ and all $h \in H_n$.

(2) One $(2^n - 1)$-dimensional representation γ which is induced by any nontrivial linear character ϕ of F_n:

$$\gamma = \phi \uparrow G_n.$$

Note that the restriction $\gamma \downarrow F_n$ of γ to F_n *equals* the direct sum of all nontrivial linear characters of F_n:

$$\gamma \downarrow F_n = \bigoplus_{1 \neq \psi \in X(F_n)} \psi.$$

Now we can state the main result of this section.

THEOREM 5. *For the groups G_n defined above,*

$$L_2(G_n) < \left(\frac{1}{2} + \frac{5}{2^{n-1}}\right)|G_n| \log |G_n|.$$

In particular, $L_2(G_n) < 0.6|G_n| \log |G_n|$ for all $n \geqq 7$.

Proof. Given $a=\sum_{g\in G_n} a_g g \in \mathbb{C}G_n$, we have to compute $\chi_1(a), \cdots, \chi_{2^n-1}(a)$ and $\gamma(a)$.

In order to compute $\gamma(a)$, write $a=\sum_{h\in H_n} \alpha_h h$ with $\alpha_h := \sum_{f\in F_n} a_{fh} f \in \mathbb{C}F_n$. Then

$$\gamma(a) = \sum_{h\in H_n} (\gamma \downarrow F_n)(\alpha_h) \cdot \gamma(h)$$

$$= \sum_{h\in H_n} \left(\bigoplus_{1\neq \psi \in X(F_n)} \psi(\alpha_h) \right) \cdot (\phi \uparrow G_n)(h).$$

According to the last formula we first compute for each $h \in H_n$ all $\psi(\alpha_h)$, $\psi \in X(F_n)$, by $|H_n|$ evaluations of a $DFT(F_n)$. This can be done by fast Hadamard–Walsh transforms in at most

$$|H_n| \cdot |F_n| \log |F_n|$$

arithmetic operations. The matrices $(\phi \uparrow G_n)(h)$ are monomial with nonzero entries equal to ± 1. (Observe that $\phi(F_n) = \{\pm 1\}$ because F_n is an elementary abelian 2-group.) The multiplication of the diagonal matrices $(\bigoplus_{1\neq\psi} \psi(\alpha_h))$ by $(\phi \uparrow G_n)(h)$ is therefore free in our computational model. Moreover, the concluding summation is also free since all the summands have their nonzero entries at pairwise disjoint positions: as γ is irreducible we have $\dim \gamma(\mathbb{C}G) = |H_n|^2$. On the other hand, the summand corresponding to h has its $\leqq |H_n|$ nonzero entries at the support of the monomial matrix $(\phi \uparrow G_n)(h)$.

To evaluate all $\chi_i(a)$ simultaneously we use the coefficients

$$b_h := \sum_{f\in F_n} a_{fh} = 1_{F_n}(\alpha_h),$$

already computed in the first step. According to (1),

$$\chi_i(a) = \eta_i \left(\sum_{h\in H_n} b_h h \right),$$

and we obtain all $\chi_i(a)$ by a single $DFT(H_n)$. Thus we get $\chi_1(a), \cdots, \chi_{2^n-1}(a)$ with at most $L_2(H_n)$ operations. Altogether we have

$$L_2(G_n) \leqq |H_n||F_n| \log |F_n| + L_2(H_n)$$

$$\leqq |G_n| \log |F_n| + 8|H_n| \log |H_n|$$

$$< \left(n + \frac{8n}{2^n} \right) |G_n|.$$

As $\log |G_n| \geqq 2n - 2^{1-n}$ for $n \geqq 2$, we get

$$L_2(G_n) < \left(\frac{1}{2} + \frac{5}{2^{n-1}} \right) |G_n| \log |G_n|. \qquad \square$$

Comparing upper and lower bounds for the groups G_n:

$$\frac{1}{4}\left(1+\frac{1}{2^n-1}\right)|G_n| \log |G_n| < L_2(G_n) < \frac{1}{2}\left(1+\frac{5}{2^{n-2}}\right)|G_n| \log |G_n|,$$

we see that they asymptotically differ by a factor of two. This is quite similar to the situation for elementary abelian 2-groups G, where we have

$$\tfrac{1}{2}|G| \log |G| \leqq L_2(G) \leqq |G| \log |G|.$$

In the next section, we will present an infinite class of groups G satisfying

$$\frac{3}{8}|G| \log |G| \leqq L_2(G) \leqq \frac{3}{4}\left(1+\frac{2}{\log |G|}\right)|G| \log |G|.$$

Again, lower and upper bounds asymptotically differ by a factor of two.

5. FFT for extra-special 2-groups. In this section, we are going to present another class of finite groups with faster Fourier transforms than those of elementary abelian 2-groups.

Let G be an extra-special 2-group of order 2^{2m+1}, i.e., the center of G is of order two and equals the Frattini subgroup of G. Up to equivalence, G has exactly the following irreducible representations (see, e.g., [13, V, 16.14]):

(1) 2^{2m} one-dimensional representations $\chi_1, \cdots, \chi_{2^{2m}}$.

(2) One 2^m-dimensional representation γ which is induced by a linear character ϕ_1 of a maximal abelian normal subgroup $A \trianglelefteq G$. Note that $|A| = 2^{m+1}$ and either $A \simeq C_4 \times C_2^{m-1}$ or $A \simeq C_2^{m+1}$ (see, e.g., [13, III, 13.8]). Moreover, $\gamma \downarrow A = \bigoplus_{i=1}^{2^m} \phi_i$, where the ϕ_i are distinct linear characters of A.

Thus like the groups G_n in the previous section, extra-special 2-groups have only one irreducible representation of large degree ($\approx\sqrt{|G|}$); all other irreducible representations are one-dimensional. Again, this situation leads to very fast Fourier transforms as follows.

THEOREM 6. *For an extra-special* 2-*group* G,

$$\tfrac{3}{8}|G| \log |G| \leqq L_2(G) < \tfrac{3}{4}|G| \log |G| + \tfrac{3}{2}|G|.$$

Proof. We have to evaluate $\chi_1(a), \cdots, \chi_{2^{2m}}(a)$ and $\gamma(a)$ for a given $a \in \mathbb{C}G$. As before, write $a = \sum_{h \in G/A} \alpha_h h$ with $\alpha_h \in \mathbb{C}A$ and evaluate the (unique) Fourier transform W_A of the abelian group A at all α_h. This takes at most $[G:A]L_2(A)$ linear operations.

Now we can compute $\gamma(a)$ according to the equation

$$\gamma(a) = \sum_{h \in G/A} (\gamma \downarrow A)(\alpha_h)\gamma(h) = \sum_{h \in G/A} \left(\bigoplus_{i=1}^{2^m} \phi_i(\alpha_h)\right)(\phi_1 \uparrow G)(h).$$

The multiplication of the diagonal matrix $(\bigoplus_{i=1}^{2^m} \phi_i(\alpha_h))$ by the monomial matrix $(\phi_1 \uparrow G)(h)$ takes at most 2^m arithmetic operations. As we can assume that one of the coset representatives $h \in G/A$ equals one, and as the concluding summation is free (see the proof of Theorem 2), $\gamma(a)$ can be computed with at most $2^m(2^m - 1)$ operations.

It remains to compute $\chi_1(a), \cdots, \chi_{2^m}(a)$. To this end, we observe that any linear character ψ of G/A can be viewed as a linear character of G by composing it with the natural projection $G \to G/A$. It is well known that the linear characters of a finite group G form an abelian group $X(G)$ under pointwise multiplication, the so-called character group of G. Thus, if χ is a linear character of G and $\psi_1, \cdots, \psi_{2^m}$ are all linear characters of G/A, then $\chi\psi_1, \cdots, \chi\psi_{2^m} \in X(G)$ are pairwise distinct and $\chi\psi_i \downarrow A = \chi \downarrow A$. By Frobenius reciprocity, the $\chi\psi_i$ are all linear characters of G whose restriction to A equals $\chi \downarrow A$. As

$$\chi\psi_i(a) = \sum_{h \in G/A} \chi\psi_i(\alpha_h) \cdot \chi\psi_i(h) = \sum_{h \in G/A} \psi_i(hA)((\chi \downarrow A)(\alpha_h) \cdot \chi(h)),$$

$\chi\psi_1(a), \cdots, \chi\psi_{2^m}(a)$ can be computed from the $(\chi \downarrow A)(\alpha_h)$ by a *DFT* of the elementary abelian 2-group G/A and $[G:A]-1$ additional multiplications. To evaluate all linear characters of G, we repeat this process 2^m times. This takes at most $2^m(L_2(G/A)+[G:A]-1)$ operations. Altogether, we have

$$L_2(G) \leqq [G:A]L_2(A)+2^m(2^m-1)+2^m(L_2(G/A)+[G:A]-1).$$

For $A \simeq C_4 \times C_2^{m-1}$, $L_2(A) \leqq 9 \cdot 2^{m-1}+4(m-1)2^{m-1}$, and

$$\begin{aligned} L_2(G) &\leqq 2^m(9 \cdot 2^{m-1}+4(m-1)2^{m-1}+2^m-1+m2^m+2^m-1) \\ &< 2^{2m+1}\left(\frac{9}{4}+\frac{3}{2}m\right) \\ &= \frac{6m+9}{4(2m+1)}(2m+1)2^{2m+1} \\ &= \tfrac{3}{4}|G| \log |G| + \tfrac{3}{2}|G|. \end{aligned}$$

For $A \simeq C_2^{m+1}$, we obtain the slightly better bound

$$L_2(G) < \tfrac{3}{4}|G| \log |G| + \tfrac{3}{4}|G|.$$

The lower bound on $L_2(G)$ follows directly from Corollary 3. □

REFERENCES

[1] M. D. ATKINSON, *The complexity of group algebra computations*, Theoret. Comput. Sci., 5 (1977), pp. 205-209.

[2] U. BAUM, M. CLAUSEN, AND B. TIETZ, *Improved upper complexity bounds for the discrete Fourier transform*, Res. Report, Dept. of Computer Science, Universität Bonn, Bonn, FRG, 1990.

[3] T. BETH, *Verfahren der schnellen Fourier-Transformation*, Teubner, Stuttgart, 1984.

[4] ———, *On the computational complexity of the general discrete Fourier transform*, Theoret. Comput. Sci., 51 (1987), pp. 331-339.

[5] L. I. BLUESTEIN, *A linear filtering approach to the computation of the discrete Fourier transform*, IEEE Trans. Automat. Control, 18 (1970), pp. 451-455.

[6] M. CLAUSEN, *Fast Fourier transforms for metabelian groups*, SIAM J. Comput., 18 (1989), pp. 584-593.

[7] ———, *Fast generalized Fourier transforms*, Theoret. Comput. Sci., 67 (1989), pp. 55-63.

[8] M. CLAUSEN AND D. GOLLMANN, *Spectral transforms for symmetric groups—fast algorithms and VLSI architectures*, in Proc. 3rd Internat. Workshop on Spectral Techniques, University of Dortmund, FRG, October 1988.

[9] J. W. COOLEY AND J. W. TUKEY, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp., 19 (1965), pp. 297-301.

[10] C. W. CURTIS AND I. REINER, *Representation Theory of Finite Groups and Associative Algebras*, John Wiley, New York, 1962.

[11] P. DIACONIS, *Spectral analysis for ranked data*, Ann. Statist., to appear.

[12] P. DIACONIS AND D. ROCKMORE, *Efficient computation of the Fourier transform on finite groups*, Tech. Report 292, Dept. of Statistics, Stanford University, Stanford, CA, April 1988.

[13] B. HUPPERT, *Endliche Gruppen* I, Springer-Verlag, Berlin, 1967.

[14] S. L. HURST, D. M. MILLER, AND J. C. MUZIO, *Spectral Techniques in Digital Logic*, Academic Press, New York, 1985.

[15] M. KAMINSKI, D. G. KIRKPATRICK, AND N. H. BSHOUTY, *Addition requirements for matrix and transposed matrix products*, J. Algorithms, 9 (1988), pp. 354-364.

[16] M. G. KARPOVSKY, *Fast Fourier transforms on finite non-Abelian groups*, IEEE Trans. Comput., 26/10 (1977), pp. 1028-1030.

[17] M. G. KARPOVSKY, ED., *Spectral Techniques and Fault Detection*, Academic Press, New York, 1985.

[18] J. MORGENSTERN, *Note on a lower bound of the linear complexity of the fast Fourier transform*, J. Assoc. Comput. Mach., 20 (1973), pp. 305-306.

[19] H. J. NUSSBAUMER, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, Berlin, 1981.

[20] C. M. RADER, *Discrete Fourier transform when the number of data points is prime*, Proc. IEEE, 56 (1968), pp. 1107-1108.

[21] D. ROCKMORE, *Fast Fourier analysis for Abelian group extensions*, Adv. in Appl. Math. 11 (1990), pp. 164–204.
[22] ———, *Computation of Fourier transforms on the symmetric group*, in Computers in Mathematics, E. Kaltofen and S. M. Watt, eds., Springer-Verlag, New York, 1989.
[23] S. WINOGRAD, *On computing the discrete Fourier transform*, Proc. Nat. Acad. Sci. U.S.A., 73 (1976), pp. 1005–1006.
[24] ———, *Arithmetic Complexity of Computations*, CBMS-NSF Regional Conference in Applied Mathematics 33, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1980.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 460-470, June 1991
© 1991 Society for Industrial and Applied Mathematics
004

ON VERTICAL VISIBILITY IN ARRANGEMENTS OF SEGMENTS AND THE QUEUE SIZE IN THE BENTLEY-OTTMANN LINE SWEEPING ALGORITHM*

JÁNOS PACH† AND MICHA SHARIR‡

Abstract. Let $S=\{e_1, \cdots, e_n\}$ be a collection of n (intersecting) line segments in the plane. Suppose that all segments have their right endpoints lying on the same vertical line, and that one wishes to bound the number of pairs of nonintersecting vertically visible segments that will intersect when extended to the right (e_i, e_j are *vertically visible* if there exists a vertical line segment connecting a point on e_i to a point on e_j and not meeting any other segment). It is shown that there are at most $O(n \log^2 n)$ such pairs, and only $O(n \log n)$ in the case of full rays, where the latter bound can be attained in the worst case. These results are applied to obtain similar upper and lower bounds on the maximum size of the queue in the original implementation of the Bentley-Ottmann algorithm for reporting all intersections between the segments in S, i.e., the implementation where future events are not deleted from the queue. It is also shown that, without the extra conditions on the segments in S and on the pairs of segments to be counted, the number of nonintersecting vertically visible pairs of segments is $O(n^{4/3}(\log n)^{2/3})$, and can be $\Omega(n^{4/3})$ in the worst case.

Key words. computational geometry, discrete geometry, line sweeping, line segments, arrangements, vertical visibility, extremal 0-1 matrices

AMS(MOS) subject classifications. 05C99, 51M99, 68Q25, 68R05

1. Introduction. Let $S=\{e_1, \cdots, e_n\}$ be a collection of n line segments in the plane. The classical line-sweeping algorithm of Bentley and Ottmann [1] for reporting all k intersections of the segments in S runs in time $O((n+k) \log n)$, as follows. It maintains a priority queue Q of future events, ordered by their x coordinates, each being either an endpoint of some e_i or a detected intersection between a pair of segments in S, which occurs to the right of the (vertical) sweepline l. Each intersection event between a pair e_i, $e_j \in S$ is added to Q when e_i and e_j become adjacent along l.

(We refer to this situation by calling e_i and e_j a pair of *vertically visible* segments. Formally this means that there exists a vertical line l cutting both e_i and e_j so that the vertical segment connecting these intersections is not crossed by any other segment of S.)

In the initially proposed implementation of the algorithm, events are added to Q when the combinatorial pattern of intersections of the segments in S with l changes, which occurs when l sweeps either through an endpoint of some e_i or through an intersection of a pair e_i, e_j (in other words, when l sweeps through the currently leftmost event in Q). In each such case, only a constant number of new vertically visible pairs occur along l, and for each such pair that actually intersects to the right of l, the corresponding intersection event is added to Q. Events are removed from Q only when l sweeps through them; that is, only events at the top of Q are removed.

* Received by the editors December 7, 1988; accepted for publication (in revised form) September 12, 1990. This research was supported by Office of Naval Research grant N00014-87-K-0129 and by National Science Foundation grants DCR-83-20085 and CCR-89-01484.

† Courant Institute of Mathematical Sciences, New York University, New York, New York 10012 and Mathematical Institute, Hungarian Academy of Sciences, Pf. 127, H-1364 Budapest, Hungary. The work of this author was supported by Hungarian Science Foundation grant OTKA-1814.

‡ Courant Institute of Mathematical Sciences, New York University, New York, New York 10012 and School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The work of this author was supported by grants from the U.S.-Israeli Binational Science Foundation and the Fund for Basic Research administered by the Israeli Academy of Sciences.

This strategy results in an algorithm whose running time is $O((n+k)\log n)$, where k is the total number of intersections between segments in S. The working storage of the algorithm is dominated by the maximum size of Q, which is certainly bounded by $2n+k$. Since k can be anything up to quadratic in n, this naive bound suggests the possibility that the worst-case working storage size might be as high as $\Omega(n^2)$. This has become a "folk-belief" among experts in the field, although no quadratic lower bound has ever been obtained.

To overcome this difficulty, a simple fix has been subsequently proposed by Brown [3]. In the modified algorithm, Q contains at all times only endpoints of the segments in S, plus intersection events that correspond to pairs that are *currently* adjacent along l (as a matter of fact, the fix in [3] is slightly different but achieves the same effect); this guarantees that the size of Q is always $O(n)$. This is achieved by deleting from Q every intersection event whose corresponding pair of segments are no longer vertically visible (i.e., adjacent) along l. Again, at every event swept through by l only a constant number of events have to be removed from Q, so the running time of the algorithm remains asymptotically the same. However, the number of update operations on Q is essentially doubled, and the implementation of Q becomes somewhat more complicated, as we now have to provide a DELETE operation that removes elements from anywhere in the queue.

In this paper we return to the original version of the Bentley-Ottmann algorithm (which does not employ the queue-deletion trick) and analyze the maximum possible size of the queue. We show that, contrary to the currently prevailing presumption, this size never exceeds $O(n \log^2 n)$. Furthermore, we show that when the algorithm is applied to a collection of lines, rather than segments, then the maximum queue size is at most $O(n \log n)$, and that this bound can be attained in the worst case. Thus, even though the size of Q can become slightly superlinear, it always remains near-linear, thus opening up the possibility of returning to the original version of the algorithm in practical applications, where the saving in the number of queue updating operations, as well as the simplicity of the data structure (which no longer requires DELETE operations to be performed for elements not in the top of the queue) may be significant.

We obtain these bounds by reducing our problem to another related one, which appears to be of independent interest, following an idea of Schorn [9]. Specifically, consider any fixed position of the sweepline l. What events are in the queue when l reaches that position? Each such event must correspond to a pair of segments that are vertically visible somewhere to the left of l and intersect to the right of l. Let us clip all segments at l, and retain only their portions to the left of l, and also discard any segment that does not reach l. Then the above observation implies that the current size of Q is bounded from above by the number of vertically visible pairs of clipped segments of S that do not intersect one another (to the left of l, that is), but whose extensions to the right do intersect. We denote this quantity for a given collection S by $\mu(S)$. Note that in this definition all segments in S are supposed to have their right endpoints on the same vertical line (the sweepline). It is easily seen that this reformulation of the problem involves no loss of information, in the sense that any lower bound M on $\mu(S)$ for some "vertically clipped" collection S, can be transformed into an instance of an execution of the Bentley-Ottmann algorithm in which the size of Q becomes greater than or equal to M.

We also consider a weaker variant of the problem (which has nothing to do with the Bentley-Ottmann algorithm), in which we are given an arbitrary collection of n segments and wish to estimate the number of pairs of nonintersecting vertically visible segments, dropping the condition that these pairs intersect when extended to the right

(and that the segments all have to end on the same vertical line). We show that the number of these pairs in this general case is $O(n^{4/3}(\log n)^{2/3})$, and can be $\Omega(n^{4/3})$ in the worst case. (Thus the innocent-looking extra conditions that are assumed in the Bentley-Ottmann case appear to be crucial for the resulting low storage bound.) This latter result is based on a random sampling technique, and its proof somewhat resembles the analysis given in [4].

The paper is organized as follows. Section 2 analyzes the case of lines, or, more generally, of a collection of segments all having the same x-projections (we refer to such configurations as *hammocks*). Section 3 analyzes the general case that arises in the Bentley-Ottmann algorithm when applied to any collection of segments, and § 4 studies the weaker variant of vertical visibility as mentioned above. Section 5 concludes with a discussion of our results and some open problems.

2. The case of a hammock. Let $S=\{e_1, \cdots, e_n\}$ be a collection of n segments all having the same x projection $[\xi, \eta]$. Thus their left endpoints all lie on the vertical line L: $x=\xi$, and their right endpoints lie on the line R: $x=\eta$. Suppose the segments are sorted in increasing vertical order of their left endpoints.

(Before continuing, we note that in this case we can drop the requirement that the pairs that we wish to count intersect when extended to the right. This is because any such pair will intersect when extended either to the right or to the left (assuming no pairs of parallel segments). Thus, since the case of a hammock is symmetric with respect to the left and right directions, we can assume, without loss of generality, that at least half of the pairs we count do intersect when extended to the right.)

Define an $n\times n$ 0-1 matrix M by putting $M_{ij}=1$ if e_i, e_j are a pair of nonintersecting vertically visible segments with e_i lying below e_j, and $M_{ij}=0$ otherwise (in particular, M is an upper triangular matrix).

LEMMA 1. *M does not contain a submatrix of the form*

$$\begin{matrix} \bigstar & 1 & 1 \\ 1 & \bigstar & 1 \end{matrix}$$

(where $\bigstar$ denotes any value). In other words, there do not exist two rows $a<b$ and three columns $x<y<z$ such that

$$M_{ay}=M_{az}=M_{bx}=M_{bz}=1.$$

Proof. Suppose to the contrary that M does contain such a submatrix. With a slight abuse of notation, let a, b, x, y, z also denote the corresponding segments in S. Thus $(a, y), (a, z), (b, x), (b, z)$ are all pairs of nonintersecting vertically visible segments, with a lying below y and z, and with b lying below x and z. Furthermore, denote by a_L, b_L, x_L, y_L, z_L the y coordinates of the left endpoints of these segments, and let a_R, b_R, x_R, y_R, z_R denote the y coordinates of their right endpoints. Then by definition we must have $a_L<b_L<x_L<y_L<z_L$. We next claim that a and x cannot intersect. Indeed, if they did intersect, then we would have $x_R<a_R<z_R$ (because a lies completely below z). Thus z would have to lie completely above x, which lies completely above b, so that b would not be able to see z at all, a contradiction which establishes the claim. A completely symmetric argument implies that b and y do not intersect.

Thus the upper envelope $\psi_{a,b}$ of a and b must lie completely below the lower envelope $\phi_{x,y,z}$ of x, y, and z, and any vertical visibility between a, b and x, y, z must occur between a pair of co-vertical points lying on these two respective envelopes. Consequently, each of these segments must appear along its corresponding envelope,

and the vertical order of their left endpoints imply that $\psi_{a,b}$ is attained from left to right first by b and then by a, and $\phi_{x,y,z}$ is attained first by x, then by y, and then by z. Let I_a, I_b, I_x, I_y, I_z denote the x-intervals where these segments appear along the corresponding envelope. Since b is assumed to see vertically both x and z, we must have $I_x \cap I_b \neq \emptyset$, $I_z \cap I_b \neq \emptyset$, which implies that $I_y \subset I_b$, which in turn contradicts the assumption that a sees y vertically, thus completing the proof of the lemma. □

It has recently been shown by Füredi [7] and independently by Bienstock and Györi [2] that 0–1 matrices that do not contain this pattern as a submatrix have at most $O(n \log n)$ 1's. Applying this result, we obtain Theorem 2.

THEOREM 2. *The maximum number of pairs of nonintersecting vertically visible segments in any collection S of n segments with the same x-projection is $\Theta(n \log n)$.*

Proof. The upper bound follows immediately from the combinatorial bounds just cited [2], [7]. For the lower bound we use the following recursive construction. We construct collections $\{S_r\}_{r \geqq 1}$ so that S_r has 2^r segments (all having $[0, 1]$ as their x-projection), with $K_r \geqq r \cdot 2^{r-1}$ pairs of nonintersecting vertically visible segments. S_1 is just a pair of nonintersecting, nearly parallel segments (with the same x-projection $[0, 1]$), so $K_1 = 1$, as required. Suppose S_r has already been constructed. To obtain S_{r+1} we construct two copies of S_r. One of them, S_r^1, is exactly S_r. The second copy S_r^2 is obtained by first rigidly translating S_r slightly upwards, and then by "shearing" it further upwards by leaving the left endpoints undisturbed and by moving each right endpoint upwards by the same very large distance c. c is chosen sufficiently large so that all intersections between segments of S_r^1 and segments of S_r^2 occur to the left of the leftmost intersection of any pair of segments in S_r. We take S_{r+1} to be $S_r^1 \cup S_r^2$. See Fig. 1 for an illustration.

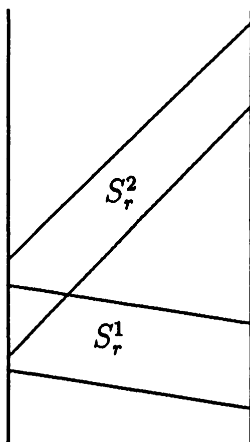


FIG. 1. *Constructing S_{r+1} from S_r.*

It is easily checked that for any $x \in [0, 1]$ and any pair of segments e_i^1, e_j^1 in S_r^1, if at x the segment e_i^1 lies above e_j^1 (respectively, lies below e_j^1, intersects e_j^1) then the same is true for the corresponding pair e_i^2, e_j^2 in S_r^2. It follows that the number of pairs of nonintersecting vertically visible segments in S_{r+1} is at least $2K_r + 2^r$, because each $e_i^1 \in S_r^1$ and the corresponding segment $e_i^2 \in S_r^2$ form a pair of nonintersecting vertically visible segments in S_{r+1}. Thus

$$K_{r+1} \geqq 2K_r + 2^r \geqq (r+1) \cdot 2^r,$$

as asserted. □

Remarks. (1) In particular, Theorem 2 implies that the queue size in the original implementation of the Bentley-Ottmann algorithm, when applied to any collection of n lines, or of n segments with the same x-projection, never exceeds $O(n \log n)$.

(2) Moreover, the lower bound construction and the observation made at the beginning of this section yield an instance of the execution of the original Bentley-Ottmann algorithm on a collection of n lines at which the queue size is $\Theta(n \log n)$.

3. The general case arising in the Bentley–Ottmann algorithm. To handle the general situation that can arise during execution of the Bentley-Ottmann algorithm on an arbitrary collection of segments, we begin by considering the following special case. Suppose S and T are two collections of n segments each, such that all segments in S have a common x-projection $[\xi, \eta]$, while each segment in T has an x-projection of the form $[\zeta, \eta]$, for some $\xi < \zeta < \eta$. We refer to segments in S as "long," and to segments in T as "short." We wish to estimate the number $\nu(S, T)$ of pairs of nonintersecting vertically visible segments (e, e') with $e \in S$, $e' \in T$, with the additional requirement that e and e' would intersect when extended to the right.

LEMMA 3. *In the above terminology, we have* $\nu(S, T) = \Theta(n \log n)$.

Proof. The lower bound follows immediately from Theorem 2. For the upper bound, define an $n \times n$ 0-1 matrix M as follows. Sort the segments in S in increasing vertical order of their left endpoints; let the resulting sequence be $s_1, \cdots, s_n$. Sort the segments in T in increasing vertical order of the intersections of the lines containing them with the line $x = \xi$, and let the resulting sequence be $t_1, \cdots, t_n$. We now put, as before, $M_{ij} = 1$ if t_i and s_j are a pair of nonintersecting vertically visible segments, whose extensions intersect to the right of $x = \eta$, and t_i lies below s_j (a symmetric analysis will handle pairs for which t_i lies above s_j). As before, we have the following claim.

CLAIM. *M does not contain a submatrix of the form*

$$\begin{matrix} \star & 1 & 1 \\ 1 & \star & 1 \end{matrix}.$$

Indeed, suppose to the contrary, that there exist segments $a, b \in T$ and $x, y, z \in S$ such that $(a, y), (a, z), (b, x), (b, z)$ are all pairs of nonintersecting vertically visible segments whose extensions intersect to the right of $x = \eta$, such that a lies below y and z, and such that b lies below x and z. Moreover, let a_R, b_R, x_R, y_R, z_R denote the y coordinates of the right endpoints of these segments, let x_L, y_L, z_L denote the y coordinates of the left endpoints of these segments, and let a_L, b_L denote the y coordinates of the intersections of the lines containing a and b with $x = \xi$. Then in the assumed configuration we have $x_L < y_L < z_L$ and $a_L < b_L$. Moreover since b and x intersect when extended to the right and b lies below x, we must also have $b_L < x_L$. Let a^*, b^* denote the extensions of a and b to the left until the line $x = \xi$ (i.e., the intersections of the lines containing a, b with the strip $\xi \leqq x \leqq \eta$). By assumption, a^* lies completely below y and z, and b^* lies completely below x and z (see Fig. 2).

As before, we claim that a^* does not intersect x, for that would make x lie completely below z, hiding it from b^*; similarly b^* does not intersect y. Thus any vertical visibility between a, b and x, y, z must be attained between their respective upper envelope $\psi_{a,b}$ and lower envelope $\phi_{x,y,z}$. Now $\phi_{x,y,z}$ behaves as before—it is attained by x, y, and z in this order from left to right along three respective intervals I_x, I_y, I_z. On the other hand, $\psi_{a,b}$ can now be attained by a, then b, and then a again (see Fig. 2), along three intervals I_{a1}, I_b, I_{a2} (where I_{a1} can be empty). But since b can see both x and z vertically, we must have $I_y \subset I_b$, so again it is impossible for a to see

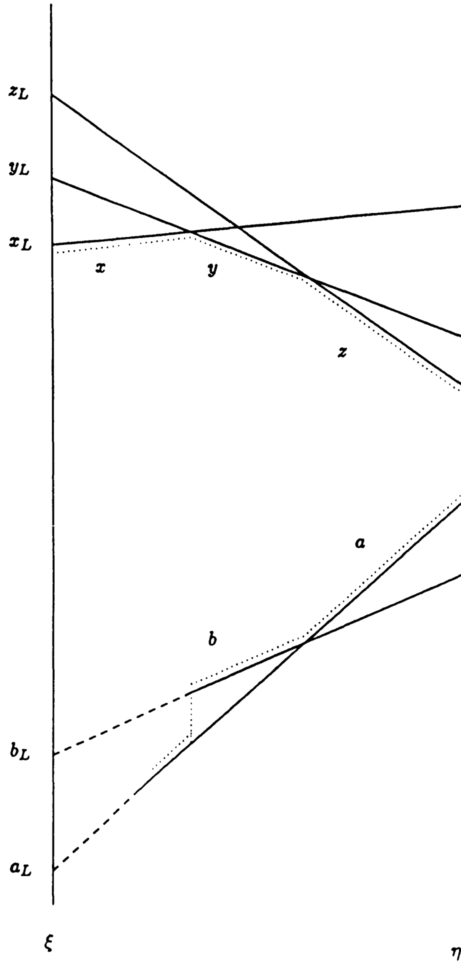


FIG. 2

y, a contradiction which completes the proof of the claim, and thus, by [7], also completes the proof of the lemma. □

THEOREM 4. *Let S be any collection of n line segments all having their right endpoints on the same vertical line. Then the number of pairs of nonintersecting vertically visible segments in S whose rightward extensions do intersect is $O(n \log^2 n)$.*

Proof. Let $\mu(S)$ denote the number of pairs of segments in S as in the theorem statement, and let μ_n denote the maximum number of such pairs for any collection S of n segments with these properties. Assume without loss of generality that the left endpoints of the segments in S have distinct x coordinates, and let x_m denote their median value. Let S_1 be the subset of roughly $n/2$ segments whose left endpoints lie to the left of x_m, and let S_2 be the complementary subset. Then we clearly have

$$\mu(S) \leqq \mu(S_1) + \mu(S_2) + \nu(S_1, S_2),$$

where $\nu(S_1, S_2)$ is the number of pairs (e_1, e_2) with $e_1 \in S_1$, and $e_2 \in S_2$ having the desired properties. By Lemma 3, this latter quantity is $O(n \log n)$, which leads to the recurrence

$$\mu_n \leqq 2\mu_{n/2} + O(n \log n),$$

which solves to $\mu_n = O(n \log^2 n)$. □

COROLLARY 5. *The maximum queue size in the original implementation of the Bentley-Ottmann algorithm, applied to any collection of n line segments, is* $O(n \log^2 n)$.

Remark. We do not know whether this bound is tight in the worst case.

4. A more general case. Although it may not be apparent from the proof of Lemma 3, it has made crucial use of the condition that the desired pairs of segments intersect when extended to the right. If we drop this condition, the number of nonintersecting vertically visible pairs can increase significantly (although still not as high as quadratic), as will be shown below.

We begin with a lower bound construction. Take an arrangement of n lines which has n faces whose total complexity is $\Theta(n^{4/3})$. Such arrangements are constructed, e.g., in [6]. The main idea in the construction is to construct a $\sqrt{n} \times \sqrt{n}$ lattice and to consider its n vertices. It is shown in [6] that one can draw $n/2$ lines, having rational slopes p/q where both p and q are small (relatively prime) integers, so that these lines have a total of $\Theta(n^{4/3})$ incidences with the lattice points. Next we modify this construction by replacing each line by a pair of parallel lines shifted by the same, arbitrarily small, distance ε. If we use the same ε for all $n/2$ lines, we obtain an arrangement of n lines, and each lattice point z becomes the "center" of a small face, whose number of bounding edges is twice the number of incidences of z with the original lines. Hence the resulting arrangement has the desired property.

For each of the n special faces f, let $\lambda(f)$, $\rho(f)$ denote, respectively, the left and right portions of its boundary, delimited by the topmost and the bottommost vertices of f (see Fig. 3).

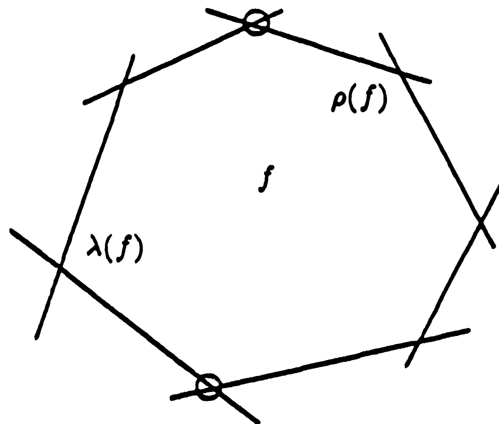


FIG. 3

Without loss of generality we can assume that the total number of edges bounding all the left portions $\lambda(f)$ of these faces is $\Theta(n^{4/3})$. Moreover, the construction in [6] also allows us to assume that the n faces in question are very small in size, so that they have pairwise disjoint y-projections. Next, for each of these faces f draw a horizontal ray r_f extending to the right from (a point slightly to the right of) the leftmost vertex of f. Let S denote the resulting collection of $2n$ lines and rays, appropriately clipped at some vertical line sufficiently distant to the right. It is clear that for each of the special faces f and for each line l appearing along $\lambda(f)$, r_f and l are nonintersecting and vertically visible in S (the former property following from the fact that no segment

r_f penetrates into another special face f'), which shows that the number of such pairs can be $\Omega(n^{4/3})$. (Note by the way that none of these pairs intersect when extended to the right.)

We next prove a closely matching upper bound, using a random sampling technique akin to that in [4]. To start the analysis we need the following variant of Lemma 3.

LEMMA 6. *Let S be a collection of n line segments, all having x-projections contained in some interval $[\xi, \eta]$, and let $m \leqq n$ be the number of "short" segments whose x-projection is not the entire $[\xi, \eta]$. Then the number of pairs of nonintersecting vertically visible segments in S is $O(mn^{1/2} + n \log n + m^{3/2}(\log m)^{1/2})$.*

Proof. Let S_1 be the subset of the m short segments and S_2 the complementary subset of "long" segments. The number of desired pairs within S_2 is $O(n \log n)$ by Lemma 1. The number of such pairs (e_1, e_2), with $e_1 \in S_1$, $e_2 \in S_2$, is analyzed as follows. Define a directed bipartite graph G between the sets S_1, S_2, which contains an edge (e_1, e_2) for every pair of nonintersecting vertically visible segments $e_1 \in S_1$, $e_2 \in S_2$, such that e_1 lies below e_2. We claim that G does not contain a copy of the complete (directed) bipartite graph $K_{2,4}$ as a subgraph. Indeed, if this were the case, there would exist two short segments a, b, and four long segments $e_1, \cdots, e_4$ such that all pairs (a, e_i), (b, e_i), $i = 1, \cdots, 4$, have the desired properties and such that both a and b lie below all four segments e_i. Let ϕ denote the lower envelope of the four e_i's and let ψ denote the upper envelope of a and b. ϕ has four intervals on the x axis so that over each of them it is attained by a fixed e_i, and ψ also has at most four such intervals so that it is attained over each of them by one of the segments a, b (see, e.g., Fig. 4).

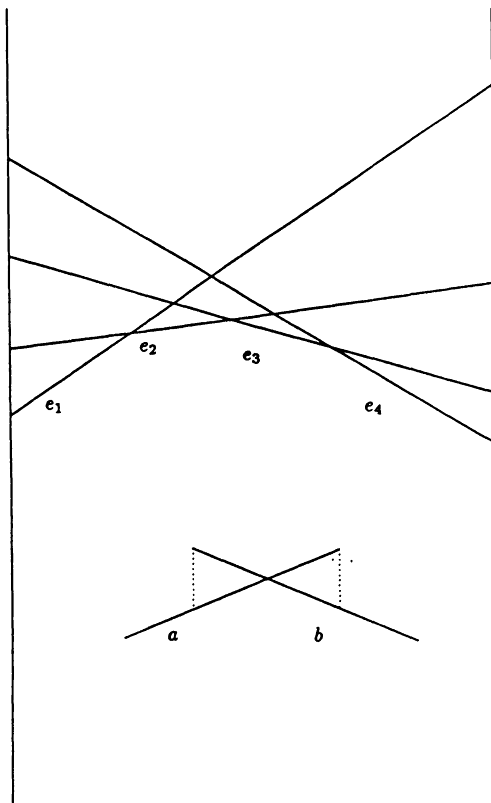


FIG. 4

By overlapping the intervals of ϕ with those of ψ and by considering all possible forms of ψ, it is easily checked that it is impossible to obtain all eight pairs of vertical visibility between a, b, and the e_i's. We can thus apply the extremal graph-theoretic lemma of Kövári, Sós, and Turán [8], which shows that a bipartite graph, whose edges connect between a set of m vertices and another set of n vertices, which does not contain $K_{2,4}$ as a subgraph, can have at most $O(mn^{1/2}+n)$ edges. Hence the number of desired pairs (e_1, e_2), with $e_1 \in S_1$, $e_2 \in S_2$, is $O(mn^{1/2}+n)$.

Finally we estimate the number of desired pairs within S_1. Assume without loss of generality that all the endpoints of the segments in S_1 lying strictly between $x=\xi$ and $x=\eta$ have distinct x coordinates. Partition the plane into $k=(m/\log m)^{1/2}$ vertical slabs $\sigma_1, \cdots, \sigma_k$ so that each of them contains at most $2(m \log m)^{1/2}$ endpoints. Consider a fixed slab σ_i, and let $p_i \leqq 2(m \log m)^{1/2}$ denote the number of segments having an endpoint in σ_i and let $q_i \leqq m$ denote the number of segments that cross σ_i all the way from left to right. The number of nonintersecting vertically visible pairs among the p_i short segments in σ_i is at most $O(p_i^2)=O(m \log m)$. The number of such pairs (e, e'), with e being short and e' being long in σ_i, is, by the preceding arguments, $O(p_i q_i^{1/2}+q_i)=O(m(\log m)^{1/2})$, and the number of such pairs among the q_i long segments is, by Lemma 1, $O(q_i \log q_i)=O(m \log m)$. Summing these bounds over all k slabs, we obtain that the total number of desired pairs within S_1 is $O(m^{3/2}(\log m)^{1/2})$. This completes the proof of the lemma. □

THEOREM 7. *The maximum number of pairs of nonintersecting vertically visible segments in any collection of n segments in the plane is* $O(n^{4/3}(\log n)^{2/3})$.

Proof. We follow the basic approach of [4], but include here, for the sake of completeness, some details of the arguments given there. Choose a random subset R of size $r=(n/\log n)^{1/3}$ of the given segments. Extend each of these segments to a full line, form the arrangement $A(R)$ of these lines, and partition its faces into $O(r^2)$ vertical trapezoidal cells, by drawing vertical segments through each intersection point until they meet another line, as in [4]. Suppose the interior of the ith cell c_i is cut by n_i original segments and contains m_i endpoints. If we clip these segments to within c_i, and apply Lemma 6, we deduce that the number of nonintersecting vertically visible pairs among these n_i clipped segments is

$$O(m_i n_i^{1/2}+n_i \log n_i+m_i^{3/2}(\log m_i)^{1/2}).$$

(Note that here we may have overestimated the global count, because we may have counted pairs of nonintersecting clipped segments, for which the full segments actually intersect.) The only pairs of nonintersecting vertically visible segments that we may have missed are those with at least one of the segments in the pair belonging to R. The contribution of each cell c_i to this extra count is easily seen to be at most $2n_i+1$, so that, summing over all cells, the number of these additional pairs is at most $O(\sum_i n_i+r^2)$.

Hence the total number of desired pairs is

$$\sum_{i=1}^{O(r^2)} O(m_i n_i^{1/2}+n_i \log n_i+m_i^{3/2}(\log m_i)^{1/2})+O(r^2).$$

Arguing as in [4], it is easy to show that $\sum_i n_i=O(nr)$. Indeed, $\sum_i n_i=\sum_{j=1}^{n} l_j$, where l_j is the number of cells crossed by the jth segment e_j. The horizon theorem for arrangements of lines (see, e.g., [5]) states that the overall complexity of all faces of $A(R)$ crossed by a line is $O(r)$. Since the number of trapezoids within a face of $A(R)$ is proportional to the complexity of the face, it easily follows that the number of trapezoids crossed by a line (or a segment) is $O(r)$; thus each $l_j=O(r)$. This establishes

the claim, which implies that

$$\sum_i n_i \log n_i = O(nr \log n).$$

The probabilistic arguments in [4] imply that there exist subsets R for which

$$\sum_i m_i n_i^{1/2} = O(m(n/r)^{1/2}),$$

where $m = \sum_i m_i \leqq 2n$, and

$$\sum_i m_i^{3/2}(\log m_i)^{1/2} \leqq \left(\sum_i m_i n_i^{1/2}\right) \cdot (\log n)^{1/2} = O(m(n/r)^{1/2}(\log n)^{1/2}).$$

Thus the total count is

$$O(m(n/r)^{1/2}(\log n)^{1/2} + nr \log n + r^2) = O(n^{4/3}(\log n)^{2/3})$$

by our choice of r. □

5. Conclusions. In this paper we have analyzed the maximum possible size of the queue in the original version of the Bentley–Ottmann line sweeping algorithm, showing that this size never exceeds $O(n \log^2 n)$ for arbitrary segments and can be at most $O(n \log n)$ in the case of lines; moreover, this latter bound can be attained in the worst case. Our solution was based on reducing the problem to a static problem analyzing the maximum number of nonintersecting vertically visible pairs of segments that do intersect when extended to the right. We have also considered a variant of this latter problem in which the "extended intersection" condition is dropped, and have shown that in this case the number of nonintersecting vertically visible pairs never exceeds $O(n^{4/3}(\log n)^{2/3})$ and can become $\Omega(n^{4/3})$ in the worst case.

The results obtained in this paper raise several open problems. One problem is whether the bound $O(n \log^2 n)$ in Theorem 4 and Corollary 5 is actually tight in the worst case, or is just an artifact of our divide-and-conquer analysis. Another problem is whether the upper bound obtained in Theorem 7 can be improved to $O(n^{4/3})$, which would then be worst-case optimal. Yet another issue is to extend our results to arrangements of more general curves. This is a natural problem since the Bentley–Ottmann algorithm also applies to such curves, and it would be nice to know that the queue size cannot become too large in these more general cases as well. Concerning this problem, we note that our results (Theorems 2 and 4) apply to collections of pseudolines or pseudosegments (namely, when the given curves are all x-monotone, and any pair of them intersects at most once).

Finally, what are the consequences of our results to pragmatic applications of the Bentley–Ottmann algorithm? Specifically, our results suggest a trade-off between the number of queue updating operations and the maximum size of the queue, and show that it is possible to save roughly half the number of updates at the cost of potentially increasing the storage for the queue by at most an $O(\log^2 n)$ factor (moreover, the implementation of the queue will be simpler, since only `INSERT` and `DELETE-MIN` operations are now required). Do these advantages justify the potentially larger storage requirements in practical executions of the algorithm?

Acknowledgment. The authors thank Peter Schorn for suggesting the problem to us and for offering the basic idea of reducing the queue analysis problem to a static one involving vertical visibility (though not quite the reduction that we have used). See [9] for more details.

REFERENCES

[1] J. Bentley and T. Ottmann, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., 28 (1979), pp. 643–647.

[2] D. Bienstock and E. Györi, *An extremal problem on* 0-1 *matrices*, SIAM J. Discrete Math., 4 (1991), pp. 17–27.

[3] K. Q. Brown, *Comments on algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., 30 (1981), pp. 147–148.

[4] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl, *Combinatorial complexity bounds for arrangements of curves and surfaces*, Discrete Comput. Geom., 5 (1990), pp. 99–160.

[5] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[6] H. Edelsbrunner and E. Welzl, *On the maximal number of edges of many faces in arrangements*, J. Combin. Theory, Ser. A, 41 (1986), pp. 159–166.

[7] Z. Füredi, *The maximum number of unit distances in a convex n-gon*, J. Combin. Theory, Ser. A, 55 (1990), pp. 316–320.

[8] T. Kövári, V. T. Sós, and P. Turán, *On a problem of K. Zarankiewicz*, Colloquium Math., 3 (1954), pp. 50–57.

[9] J. Nievergelt and P. Schorn, *Geradenprobleme mit superlinearen Wachstum*, Informatik-Spektrum, 11 (4) (1988), pp. 214–217.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 471-483, June 1991

© 1991 Society for Industrial and Applied Mathematics
005

ON POLYNOMIAL-TIME BOUNDED TRUTH-TABLE REDUCIBILITY OF NP SETS TO SPARSE SETS*

MITSUNORI OGIWARA† AND OSAMU WATANABE‡

Abstract. It is proved that if $\mathrm{P} \neq \mathrm{NP}$, then there exists a set in NP that is not polynomial-time bounded truth-table reducible (in short, $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-reducible) to any sparse set. In other words, it is proved that no sparse $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-hard set exists for NP unless $\mathrm{P} = \mathrm{NP}$. By using the technique proving this result, the intractability of several number-theoretic decision problems, i.e., decision problems defined naturally from number-theoretic problems is investigated. It is shown that for these number-theoretic decision problems, if it is not in P, then it is not $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-reducible to any sparse set.

Key words. bounded truth-table reduction, Berman–Hartmanis conjecture, sparse sets, polynomial-time hard sets, cryptography

AMS(MOS) subject classification. 68C25

1. Introduction. A set S is called *sparse* if for some polynomial p and all $n \geqq 0$, the number of elements in S of length n is less than $p(n)$. In this paper, we study the intractability of NP sets by investigating their polynomial-time reducibility to a sparse set. In particular, we consider polynomial-time bounded truth-table reducibility (in short, $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-reducibility) to sparse sets.

It has been shown that we can use "polynomial-time reducibility to a sparse set" to measure intractability of a given set [3], [9], [16]. Note that we have many kinds of polynomial-time reducibilities [8]. For each reduction type r, the concept of "$\leqq^{\mathrm{P}}_{r}$-reducibility to sparse sets" indicates a certain tractability notion. For example, a set has polynomial size circuits [10] if and only if it is $\leqq^{\mathrm{P}}_{\mathrm{T}}$-reducible (i.e., polynomial time Turing reducible) to a sparse set. A set has no p-close approximation [13], [18] unless it is $\leqq^{\mathrm{P}}_{1\text{-tt}}$-reducible (i.e., polynomial-time 1-truth-table reducible) to a sparse set.

Polynomial-time reducibility of NP sets to sparse sets has been studied by many researchers. In particular, several interesting observations have been made by considering polynomial-time many-one reducibility (in short, $\leqq^{\mathrm{P}}_{\mathrm{m}}$-reducibility), which is the most restrictive reducibility among polynomial-time reducibilities. Berman and Hartmanis [2] conjectured that every NP-complete set is not $\leqq^{\mathrm{P}}_{\mathrm{m}}$-reducible to any sparse set (here, by an NP-complete set we mean a $\leqq^{\mathrm{P}}_{\mathrm{m}}$-complete set in NP). This is equivalent to conjecturing that some NP set is not $\leqq^{\mathrm{P}}_{\mathrm{m}}$-reducible to any sparse set. Fortune [5] observed that if $\mathrm{P} \neq \mathrm{NP}$, then there exists a set in NP that is not $\leqq^{\mathrm{P}}_{\mathrm{m}}$-reducible to any co-sparse set (a co-sparse set is a set whose complement is sparse). Mahaney, improving Fortune's technique, finally obtained the following answer to the above conjecture: if $\mathrm{P} \neq \mathrm{NP}$, then no NP-complete set is $\leqq^{\mathrm{P}}_{\mathrm{m}}$-reducible to any sparse set.

On the other hand, for more general types of polynomial-time reducibility, a theorem corresponding to that of Mahaney has been left unproven. Karp and Lipton [7] showed that no NP-complete set is $\leqq^{\mathrm{P}}_{\mathrm{T}}$-reducible to any sparse set from an assumption that the polynomial-time hierarchy [14] does not collapse. However, it has been left open to show the same result from a weaker assumption that $\mathrm{P} \neq \mathrm{NP}$. Also,

* Received by the editors December 18, 1989; accepted for publication (in revised form) February 16, 1990.

† Department of Information Sciences, Tokyo Institute of Technology, Ookayama, Tokyo 152, Japan.
The research of this author was partially supported by International Information Science Foundation grant 9012219.

‡ Department of Computer Science, Tokyo Institute of Technology, Ookayama, Tokyo 152, Japan.

oracle evidence suggests that regarding Karp and Lipton, a weaker assumption will not work [6]. For $\leqq^{P}_{btt}$-reducibility, we have the following observations. Yesha [18] proved that if $P \neq NP$, then no NP-complete set is positive $\leqq^{P}_{btt}$-reducible to any co-sparse set. Ukkonen [15] showed that if $P \neq NP$, then no NP-complete set is $\leqq^{P}_{btt}$-reducible to any tally set (a tally set is a subset of 0^*). More recently, Watanabe [16] proved that if $R \neq NP$, then no NP-complete set is $\leqq^{P}_{1-tt}$-reducible to any sparse set. However, the following question has been left open.

QUESTION. Suppose that $P \neq NP$. Does NP have a set that is not $\leqq^{P}_{btt}$-reducible to any sparse set?

In this paper, we solve this question affirmatively.

Since NP-complete sets are typical sets in NP, NP-complete sets have been investigated to obtain an NP set that is not $\leqq^{P}_{r}$-reducible to any sparse set. However, it is not necessary at all to consider NP-complete sets. For example, Watanabe [16] used a "partial complement of a prefix set" to obtain his result. Here we introduce a "set of strings that are below some witness under the lexicographic order" (in short, a left set).

Let A be any set in NP. Then there exists a set C in P and a polynomial p such that for every $x \in \Sigma^*$,

$$x \in A \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)}[x \# w \in C].$$

Now for such C and p, a *left set* is the following set:

$$\text{Left}\,(C, p) = \{x \# w \colon w \in \{0,1\}^{p(|x|)} \wedge (\exists y \in \{0,1\}^{p(|x|)})[w \leqq y \wedge x \# y \in C]\},$$

where the order $\leqq$ is a standard lexicographic order. Note that Left (C, p) also belongs to NP and that Left $(C, p) \in$ P implies $A \in$ P.

What follows is a brief outline of our proof. It is easy to show that every left set Left (C, p) satisfies the following: for every x, and every w and w' in $\{0,1\}^{p(|x|)}$,

$$x \# w \in \text{Left}\,(C, p) \wedge w' < w \Rightarrow x \# w' \in \text{Left}\,(C, p).$$

By using this property, we first prove the following main theorem. If Left (C, p) is $\leqq^{P}_{btt}$-reducible to a sparse set, then Left (C, p) is in P. Now suppose that $P \neq NP$. Then there exists a set A in NP $-$ P. Thus, a left set Left (C, p) defined from A is not in P; hence it follows from our main theorem that Left (C, p) is not $\leqq^{P}_{btt}$-reducible to any sparse set. Since Left (C, p) itself is in NP, we obtain an NP set, i.e., Left (C, p), that is not $\leqq^{P}_{btt}$-reducible to any sparse set.

The notion of "left set" is often used when we formally study computational complexity of number-theoretic problems. For example, consider the prime factorization problem FACT.

PROBLEM FACT. For a given natural number n, compute the prime factorization of n.

Note that this is not a decision problem but an *evaluation problem* (i.e., a problem of computing function values). We often convert such an evaluation problem to a decision problem of similar complexity, and discuss the complexity of the converted problem. For example, instead of FACT, we often investigate the complexity of the following set:

$$\text{LF} = \{\langle a, n\rangle \colon (\exists m \colon a \leqq m < n)[m \text{ divides } n]\}.$$

Note that FACT is polynomial-time solvable if and only if LF is in P; thus, we may consider that LF characterizes the complexity of the problem FACT. For many number-theoretic evaluation problems, sets similar to LF are regarded as *natural* decision problems that characterize the complexity of the evaluation problems.

Note the similarity between LF and left sets. Following an argument similar to the one that proves our main theorem, we can prove that if LF is $\leqq_{\mathrm{btt}}^{\mathrm{P}}$-reducible to a sparse set, then LF is in P (thus, FACT is polynomial-time solvable). In other words, LF is not $\leqq_{\mathrm{btt}}^{\mathrm{P}}$-reducible to any sparse set unless FACT is polynomial-time solvable. We prove similar results for several number-theoretic problems.

2. Preliminaries. Throughout this paper, we fix our alphabet as $\Sigma=\{0,1\}$. By "string" we mean an element of Σ^*. For a string $x\in\Sigma^*$, $|x|$ denotes the length of x. We use λ to denote the null string. For any string x and y, let $x \# y$ denote a string that encodes the pair of x and y naturally; we assume that $|x\# y|=2(|x|+|y|+1)$.

Let $\|A\|$ denote the number of elements in A. For any set $A\subseteq\Sigma^*$ and any $n\geqq 0$, $A^{\leqq n}$ denote the set $\{x\in A: |x|\leqq n\}$. A set $S\subseteq\Sigma^*$ is called *sparse* if there exists a polynomial p such that $\|S^{\leqq n}\|\leqq p(n)$ for every $n\geqq 1$.

We define a standard lexicographic order on Σ^* in the following way. For any string x and y, x is lexicographically smaller than y (write $x<y$) if either (1) $|x|<|y|$, or (2) $|x|=|y|$ and there exists some k, $1\leqq k\leqq |x|$, such that (for all i: $1\leqq i<k)[x_i=y_i]$ and $(x_k=0\wedge y_k=1)$, where x_i (respectively, y_i) is the ith symbol of x (respectively, y). For any x and y, $x\leqq y$, if either $x<y$ or $x=y$.

For any strings $a, b\in\Sigma^*$ such that $a\leqq b$, an interval $[a,b]$ is the set $\{w: a\leqq w\leqq b\}$. Two intervals t and t' are *disjoint* if $t\cap t'=\emptyset$. Let $t=[a,b]$ and $t'=[a',b']$ be any two disjoint intervals. Note that either $b<a'$ or $b'<a$; we define $t<t'$ as $b<a'$.

We assume that the reader is familiar with basic concepts and notation in complexity theory (see [1], [2]).

In this paper we mainly consider polynomial-time bounded truth-table reducibility.

For any integer $k\geqq 0$, a *k-argument truth table* is a mapping α: $\{\text{TRUE}, \text{FALSE}\}^k\to\{\text{TRUE}, \text{FALSE}\}$. For any k-argument truth table α and any strings $y_1,\cdots,y_k$, a list $(\alpha, y_1,\cdots,y_k)$ is called a *k-argument truth table condition* (in short, (k-)tt-condition). For any $k\geqq 0$, a *k-truth-table function* is a function that maps every string to some l-tt condition (where $l\leqq k$). A tt-condition $(\alpha,y_1,\cdots,y_k)$ is *true relative to* B if

$$\alpha(\chi_B(y_1),\cdots,\chi_B(y_k))=\text{TRUE},$$

where χ_B is the characteristic function of B (i.e., $\chi_B(z)=\text{TRUE}\Leftrightarrow z\in B$).

A set A is *polynomial-time k-truth-table reducible* (in short, $\leqq_{k\text{-tt}}^{\mathrm{P}}$-reducible) to a set B if there exists a polynomial-time computable k-truth-table function f such that for every $x\in\Sigma^*$,

$$x\in A\Leftrightarrow(\alpha,y_1,\cdots,y_k)\text{ is true relative to }B,$$

where $f(x)=(\alpha,y_1,\cdots,y_k)$. A set A is *polynomial-time bounded truth-table reducible* (in short, $\leqq_{\mathrm{btt}}^{\mathrm{P}}$-reducible) to a set B if A is $\leqq_{k\text{-tt}}^{\mathrm{P}}$-reducible to B for some $k\geqq 0$. We write $A\leqq_{k\text{-tt}}^{\mathrm{P}} B$ (respectively, $A\leqq_{\mathrm{btt}}^{\mathrm{P}} B$) if A is $\leqq_{k\text{-tt}}^{\mathrm{P}}$-reducible (respectively, $\leqq_{\mathrm{btt}}^{\mathrm{P}}$-reducible) to B.

Many polynomial-time reducibilities other than the above have been introduced and studied. See [8] for the definition of those reducibilities, and see [3], [8], and [16] for the comparison of their computational power. It should be mentioned here that polynomial-time bounded truth-table reducibility is a generalization of polynomial-time many-one reducibility, and is special case of polynomial-time truth-table reducibility.

We introduce the concept of the "left set," which plays an important role in this paper. Let A be any set in P and p be any polynomial. A *left set* for A and p is the following set Left (A,p):

$$\text{Left}(A,p)=\{x\# w: |w|=p(|x|)\wedge(\exists y\geqq w: |y|=p(|x|))[x\# y\in A]\}.$$

For A and p, a *right set* Right (A, p) and a *range set* Range (A, p) are defined similarly:

$$\text{Right}(A, p) = \{x \# w: |w| = p(|x|) \wedge (\exists y \leqq w: |y| = p(|x|))[x \# y \in A]\},$$

$$\text{Range}(A, p) = \{x \# w_1 \# w_2: |w_1| = |w_2| = p(|x|) \wedge (\exists y: w_1 \leqq y \leqq w_2 \wedge |y| = p(|x|))[x \# y \in A]\}.$$

For any polynomial p, a *p-witness set* for a set L is a set C in P such that for every $x \in \Sigma^*$,

$$x \in L \Leftrightarrow (\exists w: w \in \{0, 1\}^{p(|x|)})[x \# w \in C].$$

Note that every set in NP has a p-witness set for some polynomial [17].

PROPOSITION 2.1. *Let L be any set in* NP, *and let C be any p-witness set for L. If* Left (C, p) *(respectively,* Right (C, p), Range (C, p)*) is in P, then L is in* P.

Proof. Let L, C, and p be as defined in the statement of the proposition. Since C is a p-witness for L, for every $x \in \Sigma^*$,

$$x \in L \Leftrightarrow (\exists w: w \in \{0, 1\}^{p(|x|)})[x \# w \in C].$$

Moreover, from the definition of left sets, it is easy to see that for every $x \in \Sigma^*$,

$$(\exists w: w \in \{0, 1\}^{p(|x|)})[x \# w \in C] \Leftrightarrow x \# 0^{p(|x|)} \in \text{Left}(C, p).$$

Thus, for every $x \in \Sigma^*$,

$$x \in L \Leftrightarrow x \# 0^{p(|x|)} \in \text{Left}(C, p).$$

Now assume that Left (C, p) is in P. Then, using a polynomial-time algorithm for Left (C, p), whether a given $x \in \Sigma^*$ is in L or not is decidable in polynomial time in $|x|$. Hence, if Left (C, p) is in P, L is in P.

The proof is similar for both Right (C, p) and Range (C, p) and thus omitted. □

3. The main theorem. In this section, we prove the following main theorem.

THEOREM 3.1. *Every left-set that is $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-reducible to a sparse set is in* P.

Consider any set $A_0 \in$ P and any polynomial p_0, and let L_0 be Left (A_0, p_0). Suppose that L_0 is btt-reducible to a sparse set S_0 via a $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-function f_0. We will show that L_0 is indeed in P.

In the following we develop a polynomial-time procedure that, for a given string $x \# w$ such that $|w| = p_0(|x|)$, searches a witness for $x \# w \in L_0$, i.e., a string $w' \in [w, 1^{p(|x|)}]$ such that $x \# w' \in A_0$. Recall that such w' exists if and only if $x \# w$ is in L_0. Thus, we can construct a polynomial time acceptor for L_0 by using the search procedure. The following is the outline of our search procedure:

procedure SEARCH (**input** $x \# w$);
    (we may assume that $|w| = p_0(|x|)$)
begin
    $U \leftarrow \{[w, 1^{p_0(|x|)}]\}$;
    repeat
        $T \leftarrow \bigcup_{t \in U}\{t_1, t_2: t_1$ (respectively, t_2)
        is the first (respectively, last) half of $t\}$;
    (*) $U \leftarrow$ choose up to $q_0(|x|)$ intervals from T;
        (q_0 is a polynomial defined later)
    until all the intervals in U are of width 1;

(**) search for a string w' in $\bigcup_{t\in U} t$ such that $x \# w' \in A_0$;
if w' exists **then return**(w')
else return("no witness exists")
end.

The key point of the above procedure is the way to select up to $q_0(|x|)$ intervals from T at (*). Suppose that for a given $x \# w$, the selection at (*) satisfies the following:

(1) $q_0(|x|)$ intervals are selected from T so that they keep a witness for $x \# w$ whenever T does, and

(2) The selection can be executed within polynomial time in $|x|$ and $\|T\|$. Then it is clear from (1) that the above procedure finds a witness for $x \# w$ (if it exists). Note the following facts: (i) the number of repetitions at the main loop is bounded by $p_0(|x|)$ (since the width of the largest interval of U becomes almost the half of the previous one at each repetition), and (ii) one can execute statement at (**) in polynomial time (since the number of strings in $\bigcup_{t\in U} t$ is bounded by $q_0(|x|)$). Thus, it follows from (2) that SEARCH terminates within polynomial time. Hence, if we show a selection procedure that satisfies (1) and (2), the proof is completed.

Consider any $x_0 \# w_0$ in L_0, where $|w_0| = p_0(|x_0|)$, and let it be fixed throughout the following discussion. Let $n_0 = |x_0|$. By "interval" we mean a subinterval of $[w_0, 1^{p_0(n_0)}]$. Define $w_{\max}$ to be lexicographically the largest w such that $w \in [w_0, 1^{p_0(n_0)}]$ and $x_0 \# w \in A_0$. Such $w_{\max}$ exists since $x_0 \# w_0 \in L_0$.

For any set T of disjoint intervals, a subset C of T is called a *cover* of T if $w_{\max}$ is not in any of intervals in $T - C$. In other words, C has $w_{\max}$ whenever T does. A cover C is called a *d-cover* if $\|C\| \leqq d$. For our purpose, it suffices to show that for any set T of disjoint intervals, we can construct a $q_0(n_0)$-cover of T within polynomial time in n_0 and $\|T\|$.

The polynomial q_0 is determined depending on S_0 and f_0. Since f_0 is the $\leqq^{\mathrm{P}}_{\mathrm{btt}}$-reduction, there exists a constant k_0 and a nondecreasing polynomial r_1 such that for every $z \in \Sigma^*$, the value $f_0(z)$ is of the form $(\alpha, y_1, \cdots, y_k)$, where $k \leqq k_0$ and $|y_i| \leqq r_1(|z|)$ for each i, $1 \leqq i \leqq k$. Hence, if y appears as an argument of some $f_0(x_0 \# w)$ (where $|w| = p_0(|x|)$), then $|y| \leqq r_1(2(n_0 + p_0(n_0) + 1))$ (recall that we assume $|x \# w| = 2(|x| + |w| + 1)$). Since S_0 is sparse, there exists a nondecreasing polynomial r_2 such that $\|S_0^{\leqq m}\| \leqq r_2(m)$ for all $m \geqq 0$. Thus, we have

$$\begin{aligned}
&\|\{y \in S_0 : y \text{ appears as an argument of } f(x_0 \# w) \text{ for some } w \in \Sigma^{p_0(n_0)}\}\| \\
&\leqq \|S^{\leqq r_1(2(n_0 + p_0(n_0) + 1))}\| \\
&\leqq r_2(r_1(2(n_0 + p_0(n_0) + 1))).
\end{aligned}$$

Define $l_0(n) = r_2(r_1(2(n + p_0(n) + 1))) + 1$; then the number of strings in S_0 appearing as an argument of $f(x_0 \# w)$ for some $w \in \Sigma^{p_0(n_0)}$ is less than $l_0(n_0)$. Finally, define the polynomial q_0 by

$$q_0(n) = (2^{2^{k_0}})^4 \cdot (l_0(n))^{k_0}.$$

Now our goal is to prove the following lemma.

LEMMA 3.2. *For any set T of disjoint intervals, there exists a $q_0(n_0)$-cover C. Furthermore, we can construct C from T within polynomial time in n_0 and $\|T\|$.*

We say that an interval $t = [a, b]$ is of *type* α if $f(x_0 \# a) = (\alpha, y_1, \cdots, y_k)$ for some $y_1, \cdots, y_k$. For a given set T of intervals, our strategy for obtaining its cover is as follows: (i) classify all the intervals in T in terms of their types (let T_α denote the

set of intervals in T of type α), (ii) for each T_α, obtain a cover C_α of T_α, and (iii) obtain C as the union of all C_α. The following fact guarantees this strategy.

FACT 1. *Let T_1 and T_2 be sets of intervals. If C_1 and C_2 are covers of T_1 and T_2, respectively, then $C_1 \cup C_2$ is a cover of $T_1 \cup T_2$.*

Proof. The proof is immediate since $(T_1 \cup T_2) - (C_1 \cup C_2) \subseteq (T_1 - C_1) \cup (T_2 - C_2)$. □

In the following, we develop a method to find such a cover C_α of T_α. The method for a general k_0 is rather complicated, so we first describe a few examples. For an interval $t = [a, b]$, $\tau(t)$ denotes the tt-condition $f_0(x_0 \# a)$. As we will show in Fact 2, if there are two intervals t and t' such that $t < t'$ and $\tau(t) = \tau(t')$, $w_{\max}$ is not in t. On the other hand, for any two intervals t and t', whether $\tau(t) = \tau(t')$ or not is decidable within polynomial time. Therefore, without loss of generality, we may assume that

(#) For any two disjoint intervals t and t' in T_α, $\tau(t) \neq \tau(t')$.

Example 1. Covers for the case $k_0 = 1$. There are only two truth tables appearing in tt-conditions; namely, id and $\neg$, where id (respectively, $\neg$) is the identity function (respectively, negation) of one argument. Let α be either of them. Let $v_1, \cdots, v_n$ be an enumeration of the intervals in T_α in increasing order, where $n = \|T_\alpha\|$. Furthermore, for each i, $1 \leqq i \leqq n$, let y_i be the argument of the tt-condition $\tau(v_i)$; that is, $\tau(v_i) = (\alpha, y_i)$. From the assumption (#), we have for each i and j, $1 \leqq i < j \leqq m$, $y_i \neq y_j$.

Now suppose that $\alpha = \text{id}$. Then, for each i, $1 \leqq i \leqq m$, $\tau(v_i)$ is true relative to S_0 if and only if $y_i \in S_0$. Recall that there are at most $l_0 - 1$ strings in S_0 appearing as an argument of some tt-condition $f_0(x_0 \# w)$. Therefore, at most $l_0 - 1$ tt-conditions $\tau(v_i)$ are true relative to S_0. Since the enumeration is in increasing order, if $\tau(v_i)$ is true relative to S_0, then, for all j, $1 \leqq j \leqq i$, $\tau(v_j)$ is true relative to S_0. Thus, for all $i \geqq l_0$, $\tau(v_i)$ cannot be true relative to S_0. Hence, $\{v_1, v_2, \cdots, v_{l_0}\}$ is an l_0-cover of T_α.

On the other hand, suppose that $\alpha = \neg$. Then, for each i, $1 \leqq i \leqq n$, $\tau(v_i)$ is false relative to S_0 if and only if $y_i \in S_0$. By a similar argument, we know that at most $l_0 - 1$ tt-conditions are false relative to S_0. Furthermore, since the enumeration is in increasing order, if $\tau(v_i)$ is false relative to S_0, then, for all j, $i \leqq j \leqq n$, $\tau(v_i)$ is false relative to S_0. Thus, for all i, $1 \leqq i \leqq n - l_0 + 1$, $\tau(v_i)$ is true relative to S_0. Hence, $\{v_{n-l_0+1}, \cdots, v_{n-1}, v_n\}$ is an l_0-cover of T_α. Therefore, for any α, $C_\alpha = \{v_1, \cdots, v_{l_0}, v_{n-l_0+1}, \cdots, v_{n-1}, v_n\}$ is a cover of T_α since C_α contains both of the above two covers. Since $\|C_\alpha\| \leqq 2l_0$, C_α is a $2l_0$-cover of T_α.

Example 2. *Covers for the case $k_0 = 2$.* Without loss of generality, we may assume that each 2-argument truth table is not reduced to any truth table of one argument or less. Then, there are only 10 truth tables of 2-arguments. Let σ_1 and σ_2 be the first and the second argument of such truth tables. Then the above-mentioned 10 truth tables are the following:

$$f_1: \text{id}(\sigma_1) \wedge \text{id}(\sigma_2), \quad f_2: \text{id}(\sigma_1) \wedge \neg(\sigma_2), \quad f_3: \neg(\sigma_1) \wedge \text{id}(\sigma_2), \quad f_4: \neg(\sigma_1) \wedge \neg(\sigma_2),$$

$$f_5: \text{id}(\sigma_1) \vee \text{id}(\sigma_2), \quad f_6: \text{id}(\sigma_1) \vee \neg(\sigma_2), \quad f_7: \neg(\sigma_1) \vee \text{id}(\sigma_2), \quad f_8: \neg(\sigma_1) \vee \neg(\sigma_2),$$

$$f_9: \sigma_1 \equiv \sigma_2, \text{ and } f_{10}: \sigma_1 \not\equiv \sigma_2.$$

Our method is based on the fact that these functions are divided into the following two types.

Type T. If $\alpha(\sigma_1, \sigma_2) = \text{TRUE}$, then either $\sigma_1 = \text{TRUE}$ or $\sigma_2 = \text{TRUE}$,

Type F. If $\alpha(\sigma_1, \sigma_2) = \text{FALSE}$, then either $\sigma_1 = \text{TRUE}$ or $\sigma_2 = \text{TRUE}$.

Note that f_1, f_2, f_3, f_5 and f_{10} are of type T, and f_4, f_6, f_7, f_8, and f_9 are of type F.

We develop a method for finding a cover only for the case $\alpha = f_6$ because this is a typical example. And for the other truth tables, similar methods are easily obtained from that by simple modifications.

We begin by developing methods to find covers of T_α when T_α satisfies some special properties. For simplicity, for any interval t, arg $(1, t)$ (respectively, arg $(2, t)$) denotes the first (respectively, the second) argument of the tt-condition $\tau(t)$, and Arg (t) denotes the set $\{\arg(1, t), \arg(2, t)\}$.

Case 1. There are $n(\geqq l_0)$ disjoint intervals $v_1, \cdots, v_n$ in T_α which satisfy the following:

$$(\forall i, j: 1 \leqq i < j \leqq n)[v_i < v_j \wedge \operatorname{Arg}(v_i) \cap \operatorname{Arg}(v_j) = \varnothing].$$

Since $\alpha = f_6$ is of type F, for each i, $1 \leqq i \leqq n$, if $\tau(v_i)$ is false relative to S_0, then $\operatorname{Arg}(v_i) \cap S_0 \neq \varnothing$; that is, $\arg(2, v_i)$ is in S_0. Furthermore, since the enumeration $v_1, \cdots, v_n$ is in increasing order, we have for any i, $1 \leqq i \leqq n$, if $\tau(v_i)$ is false relative to S_0, then there are at least $n - i + 1$ strings in S_0. Finally, since l_0 is a strict upper bound for the number of strings in S_0, $\tau(v_{n-l_0+1})$ must be true relative to S_0; thus, for any $i \leqq n - l_0 + 1$, $\tau(v_i)$ is true relative to S_0. Hence, the set $\{t \in T_\alpha : t \geqq v_{n-l_0+1}\}$ is a cover of T_α.

Case 2. There is a string y_0 such that for all $t \in T_\alpha$, $\arg(1, t) = y_0$; that is, $\tau(t)$ is of the form (α, y_0, z).

Let $v_1, \cdots, v_n$ be an enumeration of the elements in T_α in increasing order, where $n = \|T_\alpha\|$, and for each i, $1 \leqq n$, let $z_i = \arg(2, v_i)$, the second argument of $\tau(v_i)$. It may be impossible to know whether or not y_0 is in S_0 within polynomial time. But, whatever y_0 may be, we can easily construct a cover for T_α by simply combining two covers, one for the case $y_0 \in S_0$ and another for the case $y_0 \notin S_0$.

Suppose that $y_0 \in S_0$. Then, for all i, $1 \leqq i \leqq n$, $\tau(v_i)$ is true relative to S_0. Therefore, $\{v_1\}$ is a 1-cover of T_α, since v_1 is the largest element in T_α.

On the other hand, suppose that $y_0 \notin S_0$. Then, for all i, $1 \leqq i \leqq n$, $\tau(v_i)$ is false relative to S_0 if and only if $z_i \in S_0$. Since the enumeration is in increasing order, for each i, $1 \leqq i \leqq n$, if $\tau(v_i)$ is false relative to S_0, then, there exist at least $n - i + 1$ strings in S_0. Recall again that l_0 is a strict upper bound for the number of strings in S_0 appearing as an argument of tt-conditions. Therefore, $\{v_{n-l_0+1}, \cdots, v_{n-1}, v_n\}$ is an l_0-cover of T_α. Finally, let $C_\alpha = \{v_1, v_2, \cdots, v_{l_0}\}$. Then, C_α is an l_0-cover of T_α since C_α contains each of the above two covers.

Case 3. There is a string y_0 such that for all $t \in T_\alpha$, $\arg(2, t) = y_0$; that is, $\tau(t)$ is of the form (α, z, y_0).

Let $v_1, v_2, \cdots, v_n$ be an enumeration of the elements in T_α in increasing order, where $n = \|T_\alpha\|$, and for each i, $1 \leqq i \leqq n$, let z_i denote $\arg(1, v_i)$.

Based on the same idea as in Case 2, we combine two covers. Suppose that $y_0 \in S_0$. Then, for all i, $1 \leqq i \leqq n$, $\tau(v_i)$ is true relative to S_0 if and only if $z_i \in S_0$. Hence, by a similar argument, we know that $\{v_1, \cdots, v_{l_0}\}$ is a cover of T_α.

On the other hand, suppose that $y_0 \notin S_0$. Then, for all i, $1 \leqq i \leqq n$, $\tau(v_i)$ is true relative to S_0. Hence, by a similar argument, we have $\{v_n\}$ is a 1-cover of T_α (note that v_n is the largest element in T_α).

Therefore, the set $\{v_1, \cdots, v_{l_0}, v_n\}$ is an $l_0 + 1$-cover since it contains both of the above two covers.

Next we combine the above three methods. Suppose that there exists an increasing sequence of intervals $\{v_i\}_{i=1}^n$ in T_α which satisfies the following conditions:

(1) $(\forall i, j: 1 \leqq i < j \leqq n)[\operatorname{Arg}(v_i) \cap \operatorname{Arg}(v_j) = \varnothing]$,

(2) $(\forall t \in T_\alpha)(\exists i: 1 \leqq i \leqq n)[\operatorname{Arg}(t) \cap \operatorname{Arg}(v_i) \neq \varnothing]$, and

(3) $(\forall i\colon 1 \leqq i \leqq n)(\forall t \in T_\alpha : t \leqq v_i)(\exists j\colon 1 \leqq j \leqq i)[\mathrm{Arg}\,(t) \cap \mathrm{Arg}\,(v_j) \neq \emptyset]$
(as we will show later, it is easy to find such a sequence).

For each i, $1 \leqq i \leqq n$, let $U_i = \{t \in T_\alpha : \mathrm{Arg}\,(t) \cap \mathrm{Arg}\,(v_i) \neq \emptyset\}$. Then, for each i, $1 \leqq i \leqq n$, U_i is the union of the following sets:

$$U_1^{(i)} = \{t \in T_\alpha : \arg(1, v_i) = \arg(1, t)\},$$

$$U_2^{(i)} = \{t \in T_\alpha : \arg(1, v_i) = \arg(2, t)\},$$

$$U_3^{(i)} = \{t \in T_\alpha : \arg(2, v_i) = \arg(1, t)\},$$

$$U_4^{(i)} = \{t \in T_\alpha : \arg(2, v_i) = \arg(2, t)\}.$$

Note that either Case 2 or Case 3 holds for each $U_l^{(i)}$, $1 \leqq l \leqq 4$. Thus, we can find $2l_0$-covers $C_1^{(i)}$, $C_2^{(i)}$, $C_3^{(i)}$, and $C_4^{(i)}$ for $U_1^{(i)}$, $U_2^{(i)}$, $U_3^{(i)}$, and $U_4^{(i)}$, respectively (recall that there exists an l_0 cover and an l_0+1 cover for Case 2 and Case 3, respectively). Hence, $C_i = C_1^{(i)} \cup C_2^{(i)} \cup C_3^{(i)} \cup C_4^{(i)}$ is an $8l_0$-cover of U_i, and furthermore, from Fact 1, $C = \bigcup_{i=1}^n C_i$ is a cover of T_α because $T_\alpha = \bigcup_{i=1}^n U_i$ from the condition (2).

We claim that $E = \bigcup_{i=n-l_0+1}^n C_i$ is a cover of T_α. It is obvious that the claim holds for the case $n \leqq l_0$, because n being $\leqq l_0$ implies $E = C$. So, suppose that $n > l_0$. Since $\{v_i\}_{i=1}^n$ is an increasing sequence satisfying the condition (1), we know that Case 1 in the above is applicable for T_α. Then, we have that the set $\{t \in T_\alpha : t \geqq v_{n-l_0+1}\}$ is a cover of T_α. Note that the set is contained in $E' = \bigcup_{i=n-l_0+1}^n U_i$ since the condition (3) holds. Therefore, E' is a cover of T_α. For each U_i, we showed that there exists an $8l_0$-cover C_i. Then, from Fact 3 to be shown later, $E = \bigcup_{i=n-l_0+1}^n C_i$ is an $8l_0^2$-cover of T_α.

For a general k_0, we construct a method inductively by extending the method for the case $k_0 = 2$. We briefly sketch the construction. Assume that $k_0 > 2$ and for each $k < k_0$, a polynomial time method to find a polynomial cover is obtained. In a similar way as in Example 2, we find a sequence of intervals $\{v_i\}_{i=1}^n$ which satisfies conditions (1)–(3), where $\mathrm{Arg}\,(t)$ denotes the set of all arguments in $\tau(t)$. Consider U_i for some i, $1 \leqq i \leqq n$. For each $t \in U_i$, there exists some string y such that y appears in both $\tau(v_i)$ and $\tau(t)$. Therefore, according to the positions where y appears in $\tau(v_i)$ and $\tau(t)$, U_i is divided into $(k_0)^2$ subsets $U_{k,l}^{(i)}$, where $1 \leqq k, l \leqq k_0$. More precisely, for each k, l, $1 \leqq k, l \leqq k_0$, $U_{k,l}^{(i)}$ is the set $\{t \in U_i : \arg(k, v_i) = \arg(l, t)\}$. For these subsets, a similar argument as in Case 2 and Case 3 is applicable, so we can find a polynomial size cover by using a method for $k_0 - 1$. Finally, by using a similar argument as in Case 1, we obtain a polynomial size cover for T_α.

In the following, we consider any fixed truth table α with $k(\leqq k_0)$ arguments. We use K to denote $\{1, \cdots, k\}$.

We need some notions and notation. Let $t = [a, b]$ be any interval of type α, i.e., $f_0(x_0 \# a) = (\alpha, y_1, \cdots, y_k)$ for some $y_1, \cdots, y_k$. The list $(\alpha, y_1, \cdots, y_k)$ is called the *associated* tt-*condition* for t. Let $\tau(t)$ denote it; that is, $\tau(t) = f_0(x_0 \# a)$. For each $i \in K$, $\arg(i, t)$ denotes y_i, namely, the ith argument of $\tau(t)$. For any $P \subseteq K = \{1, \cdots, k\}$, $\mathrm{Arg}\,(P, t)$ denotes $\bigcup_{l \in P}\{\arg(l, t)\}$.

For any set T of intervals, a class $\{U_1, \cdots, U_m\}$ is called a *decomposition* of T if $\bigcup_{1 \leqq i \leqq m} U_i = T$ and for all i and j such that $1 \leqq i < j \leqq m$, $U_i \cap U_j = \emptyset$.

For any $P \subseteq K$, we say that a pair of intervals t and t' is *P-same* if $\arg(l, t) = \arg(l, t')$ for every $l \in P$. A set T of intervals is called a *set of P-same intervals* if every pair of intervals in T is P-same.

Let P be any subset of K, and let T be any set of disjoint and P-same intervals of type α. A sequence $(U_1, \cdots, U_m)$ is called a *left decomposition* of T with respect to P if it satisfies the following:

(1) $\{U_1, \cdots, U_m\}$ is a decomposition of T,

(2) There exist intervals $t_1, \cdots, t_m$ such that

(a) $(\forall i: 1 \leqq i \leqq m)[t_i \in U_i]$,

(b) $(\forall i: 1 \leqq i \leqq m)(\forall t \in U_i)[t_i \leqq t$ and $\text{Arg}(K-P, t) \cap \text{Arg}(K-P, t_i) \neq \emptyset]$, and

(c) $(\forall i, j: 1 \leqq i < j \leqq m)[t_i < t_j$ and $\text{Arg}(K-P, t_i) \cap \text{Arg}(K-P, t_j) = \emptyset]$.

The intervals $t_1, \cdots, t_m$ stated above are called *left-roots* of $U_1, \cdots, U_m$.

The notion of "right decomposition" is defined similarly by changing conditions (b) and (c) as follows:

(b′) $(\forall i: 1 \leqq i \leqq m)(\forall t \in U_i)[t_i \geqq t$ and $\text{Arg}(K-P, t) \cap \text{Arg}(K-P, t_i) \neq \emptyset]$; and

(c′) $(\forall i, j: 1 \leqq i < j \leqq m)[t_i > t_j$ and $\text{Arg}(K-P, t_i) \cap \text{Arg}(K-P, t_j) = \emptyset]$.

The intervals $t_1, \cdots, t_m$ for a right decomposition are called *right-roots* of $U_1, \cdots, U_m$.

Here we show that for a given T and P, we can construct a right (respectively, left) decomposition of T with respect to P within polynomial time in n_0 and $\|T\|$.

LEMMA 3.3. *Let P be any subset of K, and T be any set of disjoint and P-same intervals of type α. There exists a right (respectively, left) decomposition of T with respect to P. Furthermore, such a decomposition is computable within polynomial time in n_0 and $\|T\|$.*

Proof. We construct a right decomposition in the following way. Define $T_0 = T$, and for each $i \geqq 1$,

$$t_i = \max\{t: t \in T_{i-1}\},$$

$$U_i = \{t \in T_{i-1}: \text{Arg}(K-P, t_i) \cap \text{Arg}(K-P, t) \neq \emptyset\},$$

$$T_i = T_{i-1} - U_i.$$

Define n to be the first i such that $T_i = \emptyset$ (clearly, such i exists). Then $(U_1, \cdots, U_n)$ is a right decomposition. Note here that this construction is simple; thus it is easy to see that one can construct $(U_1, \cdots, U_n)$ within polynomial time in n_0 and $\|T\|$.

The proof is similar for a left decomposition. □

We first prove the following facts.

FACT 2. *For any disjoint intervals t and t', if $t < t'$ and $\tau(t) = \tau(t')$, then $w_{\max}$ is not in t.*

Proof. Let $t = [a, b]$ and $t' = [a', b']$ be disjoint intervals such that $t < t'$ and $\tau(t) = \tau(t')$. Since t and t' are disjoint, it follows from $t < t'$ that $b < a'$. Since $\tau(t) = \tau(t')$, we have that $f_0(x_0 \# a)$ is true relative to S_0 if and only if $f_0(x_0 \# a')$ is true relative to S_0 (recall that $\tau(t) = f_0(x_0 \# a)$ and $\tau(t') = f_0(x_0 \# a')$). Hence, $x_0 \# a \in L_0$ if and only if $x_0 \# a' \in L_0$; equivalently, $a \leqq w_{\max}$ if and only if $a' \leqq w_{\max}$. Now suppose that $w_{\max} \in t$. Then it follows that $a \leqq w_{\max} \leqq b < a'$ and thus $a \leqq w_{\max}$ and $a' > w_{\max}$; a contradiction. □

FACT 3. *For any sets C, D, and T of intervals, if C is a cover of D, and D is a cover of T, then C is a cover of T.*

Proof. The proof is straightforward and thus omitted. □

FACT 4. *Let P be any subset of K, and T be any set of disjoint and P-same intervals of type α. Let $(U_1, \cdots, U_m)$ and $(V_1, \cdots, V_n)$ be a left and a right decomposition of T with respect to P, respectively. If for every i, $1 \leqq i \leqq m$, C_i is a cover of U_i and for every*

j, $1 \leqq j \leqq n$, D_j is a cover of V_j, then $(\bigcup_{i=1}^{l_0} C_i) \cup (\bigcup_{j=1}^{l_0} D_j)$ is a cover of T, where l_0 denotes $l_0(n_0)$.

Proof. Let $C = \bigcup_{i=1}^{m} C_i$, $D = \bigcup_{j=1}^{n} D_j$, $C' = \bigcup_{i=1}^{l_0} C_i$, and $D' = \bigcup_{j=1}^{l_0} D_j$. Since $T = (\bigcup_{i=1}^{m} U_i) \cup (\bigcup_{j=1}^{n} V_j)$, $C \cup D$ is a cover of T. Hence, it follows from Fact 3 that if $C' \cup D'$ is a cover of $C \cup D$, then $C' \cup D'$ is a cover of T. Thus, to establish the fact, it suffices to show that $C' \cup D'$ is a cover of $C \cup D$.

Since T is a set of P-same intervals of type α, every $\tau(t)$ (such that $t \in T$) has the same truth table α and the same argument at each $l \in P$. Thus, for all $t \in T$, the same α' is constructed from α by substituting every lth ($l \in P$) argument of α with $\chi_{S_0}(\arg(l, t))$. Hence, for every $t = [a, b] \in T$,

$$\begin{aligned} x_0 \# a \in \mathrm{Left}(A_0, p_0) &\Leftrightarrow \alpha(\chi_{S_0}(\arg(1, t)), \cdots, \chi_{S_0}(\arg(k, t))) = \mathrm{TRUE} \\ &\Leftrightarrow \alpha'(\chi_{S_0}(\arg(l_1, t)), \cdots, \chi_{S_0}(\arg(l_h, t))) = \mathrm{TRUE}, \end{aligned}$$

where $\{l_1, l_2, \cdots, l_h\} = K - P$.

Now assume to the contrary that there exists $t \in (C \cup D) - (C' \cup D')$ such that $w_{\max} \in t$. Since $(U_1, \cdots, U_m)$ and $(V_1, \cdots, V_n)$ are decompositions of T, there exist some U_i and V_j both of which contain t. Since C_i (respectively, D_i), is a cover of U_i (respectively, V_j), we have $t \in C_i$ (respectively, $t \in D_j$). Thus, $l_0 < i$ and $l_0 < j$.

Let $u_1, \cdots, u_m$ be left roots of $(U_1, \cdots, U_m)$ and $v_1, \cdots, v_n$ be right roots of $(V_1, \cdots, V_n)$. Since $t \in U_i$ and $l_0 < i$ (respectively, $t \in V_j$ and $l_0 < j$), we have

$$u_1 < \cdots < u_{l_0} < t < v_{l_0} < \cdots < v_1.$$

Note that $w_{\max} \in t$; hence, for every w_1 and w_2 in $\{0, 1\}^{p(|x|)}$ such that $w_1 \leqq a \leqq b \leqq w_2$ (where $t = [a, b]$), we have $x_0 \# w_1 \in L_0$ and $x_0 \# w_2 \notin L_0$. Thus,

(1) $\quad (\forall i: 1 \leqq i \leqq l_0)[\alpha'(\chi_{S_0}(\arg(l_1, u_i)), \cdots, \chi_{S_0}(\arg(l_h, u_i))) = \mathrm{TRUE}]$, and

(2) $\quad (\forall j: 1 \leqq j \leqq l_0)[\alpha'(\chi_{S_0}(\arg(l_1, v_j)), \cdots, \chi_{S_0}(\arg(l_h, v_j))) = \mathrm{FALSE}]$.

We prove that no α' satisfies (1) and (2) at the same time. We consider two cases depending on α'.

Case 1. If $\alpha'(\sigma_1, \cdots, \sigma_h)$ is true, then there exists at least one i such that $\sigma_i = \mathrm{TRUE}$. Suppose that α' satisfies (1). Then for every i, $1 \leqq i \leqq l_0$, there exists at least one $l \in K - P$ such that $\chi_{S_0}(\arg(l, u_i)) = \mathrm{TRUE}$; that is, there is at least one element in $\mathrm{Arg}(K - P, u_i) \cap S_0$. Note that left roots $u_1, \cdots, u_n$ are defined so that $\mathrm{Arg}(K - P, u_i)$ is disjoint for each root. Thus, (1) implies that at least l_0 strings in S_0 appear as an argument of some associated tt-condition. This contradicts the definition of l_0.

Case 2. If $\alpha'(\sigma_1, \cdots, \sigma_h)$ is false, then there exists at least one i such that $\sigma_i = \mathrm{TRUE}$. Following an argument similar to the above, we can prove that α' does not satisfy (2). □

By using these facts, we can prove the following lemma.

LEMMA 3.4. *Let P be any subset of K and let T be any set of disjoint and P-same intervals of same type (say α). Then, there exists a $(h!)^2(2l_0)^h$-cover of T, where h and l_0 denote $\|K - P\|$ and $l_0(n_0)$, respectively.*

Proof. We prove the lemma by induction over h. For the induction base, consider the case $h = 0$. Since $h = 0$, P is exactly K; that is, for all t and t' in T, $\tau(t) = \tau(t')$. Let $\hat{t} = \max\{t: t \in T\}$ and $C = \{\hat{t}\}$. Then, from Fact 2, C is a 1-cover of T.

For the inductive step, let $h > 0$ and suppose that the claim holds for all h' less than h. Let $(U_1, \cdots, U_m)$ be a left decomposition of T with respect to P and $u_1, \cdots, u_m$ be left-roots of $U_1, \cdots, U_m$. Let any l, $1 \leqq l \leqq m$ be fixed here. For each i and $j \in K - P$, consider the following set:

$$U_{i,j}^{(l)} = \{t \in U_l : \arg(i, u_l) = \arg(j, t)\}.$$

Note that $U_l = \bigcup_{i,j\in K-P} U_{i,j}^{(l)}$. Note furthermore that $U_{i,j}^{(l)}$ is $P\cup\{j\}$-same; thus, for every i and $j\in K-P$, $U_{i,j}^{(l)}$ has a $((h-1)!)^2 2^{h-1} l_0^{h-1}$-cover, which we denote by $C_{i,j}^{(l)}$. Now define $C_l = \bigcup_{i,j\in K-P} C_{i,j}^{(l)}$. It follows from Fact 1 that C_l is a $(h!)^2 2^{h-1} l_0^{h-1}$-cover of U_l.

Similarly, let $(V_1, \cdots V_n)$ be a right decomposition of T with respect to P and $v_1, \cdots v_n$ be right-roots of $V_1, \cdots, V_n$. Let any l, $1\leqq l\leqq t$ be fixed here. For each i and $j\in K-P$, consider the following set:

$$V_{i,j}^{(l)} = \{t\in V_l : \arg(i, v_l) = \arg(j, t)\}.$$

Note that $U_l = \bigcup_{i,j\in K-P} U_{i,j}^{(l)}$. Then, by a similar argument to the above, we have for every i and $j\in K-P$, $V_{i,j}^{(l)}$ has a $((h-1)!)^2 2^{h-1} l_0^{h-1}$-cover. Let $D_{i,j}^{(l)}$ denote such a cover, for each i and $j\in K-P$, and define $D_l = \bigcup_{i,j\in K-P} D_{i,j}^{(l)}$. It follows from Fact 1 that D_l is a $(h!)^2 2^{h-1} l_0^{h-1}$-cover of V_l.

Finally, define $E = (\bigcup_{l=1}^{l_0} C_l)\cup(\bigcup_{l=1}^{l_0} D_l)$. It follows from Fact 4 that E is a cover of T. Note that $\|E\| \leqq 2l_0 \cdot \{(h!)^2 2^{h-1} l_0^{h-1}\} = (h!)^2 2^h l_0^h$. Therefore the lemma holds for h. □

LEMMA 3.5. *Let T be a set of disjoint intervals of type α for some k-argument truth-table α. There exists a $(k!)^2 2^k l_0^k$-cover of T, where l_0 denotes $l_0(n_0)$.*

Proof. The proof is immediate by considering the case $P=\emptyset$ in Lemma 3.4. □

From the proofs of Lemma 3.4 and 3.5, it is easy to see that the covers considered in the above lemmas are polynomial-time computable.

Now we prove Lemma 3.2 thereby completing the proof of Theorem 3.1.

Proof of Lemma 3.2. Let F be a set of all truth tables of which the number of arguments are bounded by k_0. It is clear that $\|F\| = 2^{2^{k_0}}$. For each truth table α, let T_α be the set of intervals in T of type α. Note that $T = \bigcup_{\alpha\in F} T_\alpha$. From Lemma 3.4, for each α, there exists a $(k_0!)^2 \cdot 2^{k_0} \cdot l_0^{k_0}$-cover for T_α, which we denote by C_α. Now letting $C = \bigcup_{\alpha\in F} C_\alpha$, we have that C is a cover of T from Fact 1 such that $\|C\| \leqq 2^{2^{k_0}}(k_0!)^2 2^{k_0} \cdot l_0(n_0)^{k_0} \leqq (2^{2^{k_0}})^4 l_0(n_0)^{k_0} = q_0(n_0)$. Hence, there exists a $q_0(n_0)$-cover for T.

Furthermore, it is easy to see that computing T_α for a given T and α can be executed in polynomial time in n_0 and $\|T\|$. Therefore, a $q_0(n_0)$-cover can be computed within polynomial time in n_0 and $\|T\|$. □

Similar results are also available for right sets and range sets.

THEOREM 3.6. *Every right set that is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set is in* P.

THEOREM 3.7. *Every range set that is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set is in* P.

4. Bounded truth-table reducibilities of NP sets. In this section, we consider the intractability of NP sets.

In the previous section, we showed that for every set C in P and for every polynomial p, if Left (C, p) is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set, Left (C, p) is in P. On the other hand, Proposition 2.1 states that if Left (C, p) is in P, the NP set for which C is a p-witness is already in P. Combining these two results, we immediately have the following corollary.

COROLLARY 4.1. *Let L be a set in* NP, *C be a set in* P, *and p be a polynomial such that C is a p-witness for L. If* Left (C, p) *is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set, then L is in* P.

Similar results are also available for right sets and range sets.

COROLLARY 4.2. *Let L be a set in* NP, *C be a set in* P, *and p be a polynomial such that C is a p-witness for L. If* Right (C, p) *is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set, then L is in* P.

COROLLARY 4.3. *Let L be a set in* NP, *C be a set in* P, *and p be a polynomial such that C is a p-witness for L. If* Range (C, p) *is $\leqq_{\rm btt}^{\rm P}$-reducible to a sparse set, then L is in* P.

From any of the above three corollaries, we can easily obtain the following corollary.

COROLLARY 4.4. *If* $P \neq NP$, *then there exists a set in* NP *that is not* $\leqq_{btt}^{P}$*-reducible to any sparse set.*

Proof. The proof is straightforward and is thus omitted. □

Next we consider the complexity of number-theoretic problems. We define some problems.

(1) FACT: for a natural number n, compute a prime factorization of n.

(2) DISC: for natural numbers n, a, and b, compute the minimum $l > 0$ such that $a \equiv b^l \pmod{n}$ if it is defined (such l is called "the discrete logarithm of a base b modulo n").

(3) SQRT: for natural numbers n and a, compute one of the square roots of a modulo n if they exist.

These problems are deeply related to some cryptosystems. ElGamal's cryptosystem [4], and Rabin's cryptosystem [11] are based on the difficulty of solving DISC and SQRT, respectively. FACT is the most fundamental problem in number theory and it is known that if FACT is easily solvable, then SQRT is easily solvable. Also, if SQRT $\in$ P, then FACT can be done in probabilistic polynomial time.

Since these problems are evaluation problems, we often convert them to decision problems of similar complexity. For example, consider the following sets in Σ^*:

(1) $LF = \{\langle a, n\rangle : (\exists m : a \leqq m < n)[m \text{ divides } n]\}$,

(2) $RD = \{\langle a, b, k, n\rangle : (\exists m : 0 < m \leqq k)[a \equiv b^m \pmod{n}]\}$,

(3) $RGS = \{\langle a, u, v, n\rangle : (\exists m : u \leqq m \leqq v)[a \equiv m^2 \pmod{n}]\}$,

where $\langle \cdot, \cdot \rangle$ is a pairing function of integers.

It is easy to see that these sets are in NP, and each of them has computational complexity similar to the corresponding evaluation problem.

Following the proof of our main theorem, we can prove the similar results for these problems.

COROLLARY 4.5. *If* LF (*respectively,* RD, RGS) *is* btt*-reducible to a sparse set, then* LF (*respectively,* RD, RGS) *is in* P, *and consequently* FACT (*respectively,* DISC, SQRT) *is polynomial time solvable.*

Acknowledgments. The authors thank Professor Kojiro Kobayashi for his encouragement and useful discussions with him. They are also grateful to Richard Beigel and an anonymous referee for their valuable suggestions and for detecting errors in the manuscript.

REFERENCES

[1] J. L. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity* I, EATCS Monogr. Theoret. Comput. Sci., No. 11, Springer-Verlag, Berlin, New York, 1988.

[2] L. BERMAN AND J. HARTMANIS, *On isomorphism and density of* NP *and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[3] R. V. BOOK AND K. KO, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[4] T. ELGAMAL, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Inform. Theory, 31 (1985), pp. 469–472.

[5] S. FORTUNE, *A note on sparse complete sets*, SIAM J. Comput., 8 (1979), pp. 431–433.

[6] N. IMMERMAN AND S. MAHANEY, *Relativising relativized computations*, Theoret. Comput. Sci., 68 (1989), pp. 267–276.

[7] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1980, pp. 302–309.

[8] R. LADNER, N. LYNCH, AND A. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.

[9] S. MAHANEY, *Sparse complete sets for* NP: *Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[10] N. PIPPENGER, *On simultaneous resource bound*, in Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, Institute of Electrical and Electronics Engineers, CA, 1979, pp. 307–311.

[11] M. RABIN, *Digitalized signatures and public-key function as intractable as factorization*, Tech. Report MIT/LCS/TR-212, Massachusetts Institute of Technology, Cambridge, MA, 1979.

[12] U. SCHÖNING, *Complexity and Structure*, Lecture Notes in Computer Science 211, Springer-Verlag, Berlin, New York, 1985.

[13] ———, *Complete sets and closeness to complexity classes*, Math. Systems Theory, 19 (1986), pp. 29–41.

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

[15] E. UKKONEN, *Two results on polynomial time truth-table reductions to sparse sets*, SIAM J. Comput., 12 (1983), pp. 580–587.

[16] O. WATANABE, *On* $\leqq^{P}_{1\text{-tt}}$ *sparseness and nondeterministic complexity classes, automata, language and programming*, Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, New York, 1988, pp. 697–709.

[17] C. WRATHALL, *Complete-sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23–33.

[18] 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. 20, No. 3, pp. 484-498, June 1991
© 1991 Society for Industrial and Applied Mathematics
006

NONDETERMINISTIC COMPUTATIONS IN SUBLOGARITHMIC SPACE AND SPACE CONSTRUCTIBILITY*

VILIAM GEFFERT†

Abstract. The open problem of nondeterministic space constructibility for sublogarithmic functions is resolved. It is shown that there are no unbounded monotone increasing nondeterministically space constructible functions such that $\sup_{n\to\infty} s(n)/\log(n)=0$. Consequently, space constructibility of monotone functions cannot be used to separate nondeterministic space from deterministic space, even for a very low-level complexity range, since functions like $\log\log(n)$ and $\sqrt{\log(n)}$ are not space constructible by nondeterministic Turing machines. This result is obtained by the extension of the $n\to n+n!$ method, described in [*Hierarchies of memory limited computations*, IEEE Conference Record on Switching Circuit Theory and Logical Design, 1965, pp. 179–190], to the nondeterministic case.

Key words. space-bounded computation, space constructibility, nondeterministic Turing machine, nondeterministic space

AMS(MOS) subject classifications. 68Q15, 68Q75, 68Q05

1. Introduction. One of the important open problems in space complexity theory is the existence of nondeterministically space-constructible functions growing faster than $\log\log(n)$, but not as fast as $\log(n)$. Two of the most important results in space complexity theory, Savitch and Immerman–Szelepcsényi theorems for $s(n)$ space-bounded nondeterministic Turing machines, were proved only for $s(n)\geqq\log(n)$ ([7], [5], [11]).

It was shown ([1], [8], [3]) that no unbounded monotone increasing function below $\log(n)$ is fully space constructible by the deterministic machines. The best-known lower bound for the nondeterministic case was $\log\log(n)$.

It was also shown [3] that we could separate nondeterministic space from deterministic space, by showing a nondeterministically fully space-constructible function below $\log(n)$. In particular, if $\log\log(n)$ were fully space constructible by a nondeterministic Turing machine, then we would have $\text{SPACE}(s(n))\subsetneq\text{NSPACE}(s(n))$, for any $s(n)$ between $\log\log(n)$ and $\log(n)$.

We shall show that no unbounded monotone increasing function with $\sup_{n\to\infty} s(n)/\log(n)=0$ can be fully space constructible by nondeterministic Turing machines. Since $\log(n)$ is space constructible (even deterministically), this lower bound is optimal. We shall prove it by showing that if a nondeterministic machine A traverses the whole input (from margin to margin), then A can do it by a computation path which repeats, on a substantial portion of the input tape, some loop in a very deterministic way. For such computation paths, we can use the $n\to n+n!$ method, which was used to prove a similar result for deterministic machines ([10], [1], [8]).

The paper is organised as follows: We begin in § 2 by giving some basic definitions. Section 3 develops some properties of nondeterministic computations in sublogarithmic space, shows the "pseudodeterministic" behavior of some computation paths, and then proves the main result of the paper. Section 4 discusses some consequences of the main "Constructibility Theorem" and resolves some open problems.

2. Preliminaries. We shall deal with the standard Turing machine model, which has a finite control, a two-way read-only input tape, and a separate semi-infinite

---

* Received by the editors February 12, 1990; accepted for publication (in revised form) June 6, 1990. This work was supported by State Plan of Fundamental Research grant I-1-5/08.

† Department of Computer Science, University of P. J. Šafárik, Jesenná 5, 04154 Košice, Czechoslovakia.

two-way read-write worktape. (See [4], [6], or [10] for the exact definition and properties of such machines.)

DEFINITION 1. *A memory state* of a Turing machine is an ordered triple $k = (q, u, i)$, where q is a state of the machine's finite control, u is a string of worktape symbols written on the worktape (not including the left endmarker or blank symbols), and i is a position of the worktape head ($0 \leqq i \leqq |u|+1$, where $|u|$ denotes the length of u).

Note that the position of the input tape head has not been incorporated into this definition and will always be displayed separately.

DEFINITION 2. *The size of a memory state* $k = (q, u, i)$ is the length of the worktape space used, i.e., $|u|$. We shall denote it by $/k/$. The size of the initial memory state k_I is zero.

We may assume, without loss of generality, that our Turing machine is not allowed to write the blank symbol on the worktape or reduce the size of its memory state. Therefore, we may assume that if a memory state k_2 can be reached from k_1 by some computation path, then $/k_2/ \geqq /k_1/$. This modification of the Turing machine does not increase its space complexity.

DEFINITION 3 [3]. A function $s(n)$ is *fully space constructible* if there exists a Turing machine which for all inputs of length n marks off $s(n)$ space on its worktape and stops, using no more than $s(n)$ space.

DEFINITION 4 [3]. A function $s(n)$ is *fully space constructible nondeterministically*, if there is a nondeterministic Turing machine which is $s(n)$ space bounded on all computation paths and uses exactly $s(n)$ space on at least one computation path on every input of length n.

Before passing to our main result, one pathological case should be eliminated: There is no loss of generality in assuming that

$$s(n) \geqq 1 \quad \text{for each } n, \tag{1}$$

since, if $s(n)$ is fully space constructible in sublogarithmic space, then so is $s'(n) = s(n)+1$. Therefore, we shall not consider functions with $s(n) = 0$, for any n.

3. Sublogarithmic space constructibility. We can now state and prove the main result. We show that if $s(n)$ is nondeterministically space constructible and $\sup_{n\to\infty} s(n)/\log(n) = 0$, then there exists $\check{n}$ such that for each $n \geqq \check{n}$ we have

$$s(n) = s(n+n!) = s(n+2n!) = s(n+3n!) = \cdots.$$

Before doing this, we need some technical lemmas, which follow.

LEMMA 1 (Number of Memory States). *For each $s(n)$ space-bounded nondeterministic Turing machine, there exists a constant $c \geqq 6$ such that the number of reachable memory states for all inputs of length n is at most $c^{s(n)}$, for each n.*
(The argument is obvious: we use only (1) and the fact that the position of the input tape head is not a part of the memory state.) Thus, we have a constant c such that

$$\text{No. of memory states} \leqq c^{s(n)} \quad \text{for each } n, \qquad 6 \leqq c. \tag{2}$$

LEMMA 2 (Sublogarithmic $s(n)$). *For each $s(n)$ such that* $\sup_{n\to\infty} s(n)/\log(n) = 0$, *and each $c \geqq 6$, there exists $\check{n} \geqq 2$ such that*

$$(c^{s(n)})^6 < n, \qquad 1 \leqq c^{s(n)}/6 \quad \text{for each } n \geqq \check{n}. \tag{3}$$

(We need (1) to show that $c^{s(n)}/6 \geqq 1$. The rest of the argument is straightforward, since there can be only finitely many n's such that $s(n)/\log(n) \geqq 1/6 \log(c)$.)

Now, let $s(n)$ be a nondeterministically fully space-constructible function with $\sup_{n\to\infty} s(n)/\log(n) = 0$, and let A be the nondeterministic $s(n)$ space-bounded Turing machine constructing $s(n)$. Then, because of Lemmas 1 and 2, we have $c \geqq 6$ and $\check{n} \geqq 2$ such that $(c^{s(n)})^6 < n$ and $1 \leqq c^{s(n)}/6$, for each $n \geqq \check{n}$. Define

$\check{s} = s(\check{n})$—the maximum size of memory states reachable from the initial memory state, for all inputs of length $\check{n}$,

$M = c^{s(\check{n})}$—an upper bound on the number of reachable memory states, for all inputs of length $\check{n}$.

Clearly, by Lemmas 1 and 2, we have that $M^6 < \check{n}$, $1 \leqq M/6$, and therefore

$$1 \leqq \tfrac{1}{6}M \leqq M \leqq \tfrac{1}{6}M^2 \leqq M^2 \leqq \tfrac{1}{6}M^3 \leqq M^3 \leqq \cdots \leqq M^6 < \check{n} < \check{n} + \check{n}!. \tag{4}$$

Because the machine A must mark off $s(n)$ space for all inputs of length n, it is sufficient to consider only inputs of the form 1^n, where $n \geqq \check{n}$. From now on, we shall concentrate on the inputs $1^{\check{n}}$ and $1^{\check{n}+\check{n}!}$. However, the argument can be easily extended for each $n \geqq \check{n}$, because (3) is valid for each $n \geqq \check{n}$.

In what follows, unless otherwise stated, we shall assume that the input word is $1^{\check{n}+\check{n}!}$.

The following lemma asserts, for input $1^{\check{n}+\check{n}!}$ and until the space used has not exceeded $\check{s} = s(\check{n})$, that it is possible to replace each computation path making a "long U-turn" (begin and end at the same position of the input tape visiting neither of the endmarkers) by an equivalent computation path using a "short-cut," i.e., by a computation not moving the input head more than M^2 positions to the right (or left).

LEMMA 3 (U-Turn). *The following holds for all memory states k_1, k_2 such that $/k_1/ \leqq /k_2/ \leqq \check{s}$: If there exists a computation path such that*

(a) *the first memory state is k_1 with the input head at a position i,*

(b) *the last memory state is k_2 at the same position,*

(c) *the input head is never moved to the left of i, nor does it visit the right endmarker, then there also exists a computation path satisfying* (a), (b), *and* (c) *such that the input head is never moved farther than M^2 positions to the right of i.* (*The same holds for computations not visiting the left endmarker, taking place to the left of i.*) (*See Fig.* 1.)

Proof. Suppose that the furthest position of the input head, in the computation path from k_1 to k_2, is $i+h$ in memory state k, where $h > M^2$.

Let p_j be the last memory state of the computation path from k_1 to k such that the input head was at the position $i+j$, for $j = 0, \cdots, M^2$. Similarly, let q_j be the first memory state of the path from k to k_2 such that the input head was back at the position $i+j$.

Clearly, the input head is never moved to the left of $i+j$ in the computation path from p_j to q_j, for each $j = 0, \cdots, M^2$.

Since $/k_2/ \leqq \check{s}$, the memory states $p_0, p_1, p_2, \cdots$ and $q_0, q_1, q_2, \cdots$ are $\check{s}$-bounded. Therefore there must be at least one pair of memory states in the sequence $(p_0, q_0), \cdots, (p_{M^2}, q_{M^2})$ which is repeated, because there are at most M different memory states not bigger than $\check{s}$. Thus we have $j_1 < j_2$ such that $(p_{j_1}, q_{j_1}) = (p_{j_2}, q_{j_2}) = (p, q)$.

But then we can remove the computation paths from p_{j_1} to p_{j_2}, and from q_{j_2} to q_{j_1}, and we still have a computation starting in k_1 at position i and ending in k_2 at the same position, because the input word consisted of identical symbols (ones) and the head never visited the right endmarker.

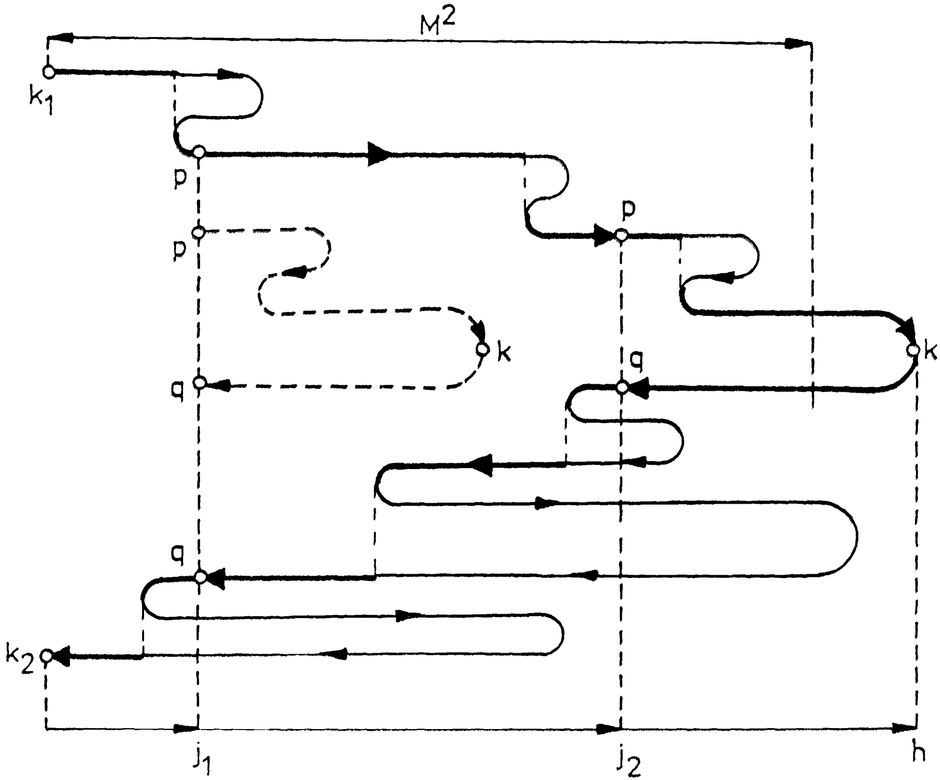


FIG. 1

Note that now the memory state k corresponds to the position $i+h-(j_2-j_1)<i+h$. This implies that the shortest possible computation path from k_1 to k_2 never moves the input head further than M^2 positions to the right. The same holds also for computations taking place to the left of i. □

The next lemma states that each computation path not using more than $\check{s}$ space is "position independent," i.e., it can be "moved" freely to any position of the input tape provided that both initial and final positions of this computation path lie outside "critical zones," that is, at least M^2+1 positions away from either margin.

LEMMA 4 (Position Independence). *The following holds for all memory states k_1, k_2 such that $/k_1/ \leqq /k_2/ \leqq \check{s}$:*

If the machine A can get from memory state k_1 to k_2 by moving its input head from position i_1 to $i_2=i_1+l$, visiting neither of the endmarkers, then A can get from k_1 to k_2 by moving its head from a position j_1 to $j_2=j_1+l$, for each j_1, j_2 such that

$$M^2+1 \leqq j_1 \leqq j_2 \leqq \check{n}+\check{n}!-(M^2+1),$$

(i.e., if both j_1 and j_2 are at least M^2+1 positions away from the margins). A similar condition can also be formulated for moving to the left. (*See Fig.* 2.)

Proof. By the U-Turn Lemma, in the shortest possible computation path from k_1 to k_2, A never moves the input head more than M^2 positions to the left of i_1, nor can the head be moved to the right of i_2+M^2. Otherwise, we could find a U-turn longer than M^2, which is not possible in the shortest computation path.

Therefore, for each j_1 and $j_2=j_1+l$ such that $M^2+1 \leqq j_1 \leqq j_2 \leqq \check{n}+\check{n}!-(M^2+1)$, we can move the whole computation path so that it starts at position j_1 and ends at

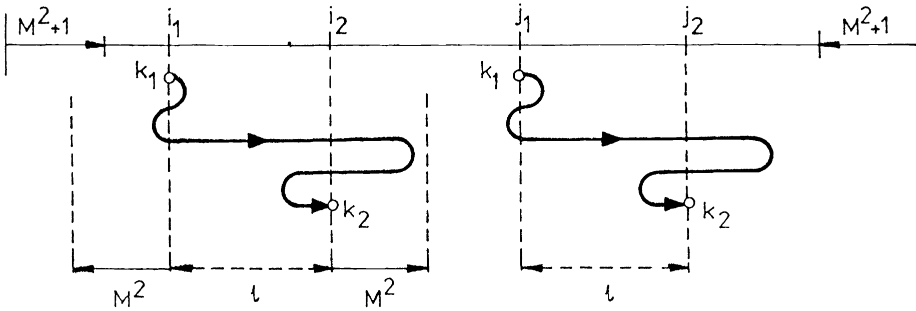


FIG. 2

$j_2 = j_1 + l$ without the risk that the input head will hit any of the margins (which could change the behavior of A). □

DEFINITION 5. *A loop of length l beginning in memory state k* is a computation starting in memory state k at position i_1 and ending in the same memory state k at position $i_2 = i_1 + l$. Moreover, neither of the endmarkers is visited by the input head during this computation.

Thus, the length of the loop is not the number of computation steps here, but rather the distance between the initial and final input head positions.

Clearly, the nondeterministic Turing machine A is far from repeating regularly any loop it gets into, because it can jump out of the loop by making a nondeterministic decision. Still, we shall show that if A traverses the whole input, then A can also traverse it by a computation regularly iterating a "short" loop (of length $l_0 \leqq M$) such that the portions of the input tape traversed before and after this iteration are of lengths s_1 and s_2, which are shorter than M^4, but longer than M^2+1. This means, roughly speaking, that s_1 and s_2 are "short," and that the whole iteration lies outside the "critical zones" near the margins.

THEOREM 1 (Dominant Loop). *The following holds for all memory states q_1, q_2 such that $/q_1/ \leqq /q_2/ \leqq \check{s}$: If the input $1^{\check{n}+\check{n}!}$ can be traversed from left to right by a computation path beginning in q_1 and ending in q_2 such that the endmarkers are visited only in q_1 and q_2, then $1^{\check{n}+\check{n}!}$ can be traversed by an equivalent computation path such that A,*

(a) *having traversed s_1 positions,*

(b) *gets into a loop of length l_0 which is repeated r_0 times,*

(c) *then traverses the rest of the tape of length s_2, for some s_1, l_0, r_0, s_2 such that*

$$1 \leqq l_0 \leqq M,$$
$$M^2+1 \leqq s_1 \leqq M^4, \tag{5}$$
$$M^2+1 \leqq s_2 \leqq M^4.$$

Proof. Since, by (4), $(M^2+1)+(M+1)+(M^2+1) < \check{n} < \check{n}+\check{n}!$, we can consider two segments of length (M^2+1) at either end of the input tape, and still have room for one segment of length $(M+1)$ placed at the left (see Fig. 3).

FIG. 3

Since there are at most M different memory states not bigger than $\check{s}$, the machine must, in each segment of length $M+1$, enter some memory state twice, i.e., it must execute a loop, beginning in some memory state k_0, of length l_0 such that $1 \leqq l_0 \leqq M$. Let s_1 be the length of the tape segment traversed before the execution of this loop (see Fig. 4), and let s_2 denote the length traversed after:

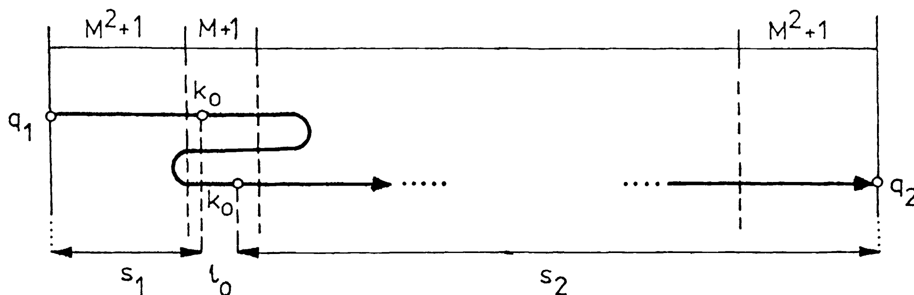


FIG. 4

Clearly, $M^2+1 \leqq s_1 \leqq (M^2+1)+(M+1)$, and $M^2+1 \leqq s_2$. But then, using (4), we have

$$1 \leqq l_0 \leqq M,$$
$$M^2+1 \leqq s_1 \leqq M^4, \tag{6}$$
$$M^2+1 \leqq s_2.$$

But we would like to have $s_2 \leqq M^4$, i.e., we would like to have a computation path with s_2 as short as possible. Therefore, among all possible computation paths starting in q_1 and ending in q_2 such that A, after traversing s_1 positions, gets into a loop of length l_0 which is repeated r_0 times, and then traverses the rest of the input of length s_2, for some s_1, l_0, r_0, s_2 satisfying (6), choose a computation path with minimal s_2. (We already have one such computation, with the number of iterations $r_0 = 1$, hence, the computation with minimal s_2 really exists.) Note that by minimizing s_2 we maximize r_0.

We shall show that, in a computation path with minimal s_2, no loop of any length l can occur more than l_0 times between positions $h_1 = s_1 + r_0 l_0$ and $h_2 = \check{n} + \check{n}! - (M^2+1)$, i.e., after leaving the iteration of l_0-loop and before entering the critical zone near the right margin. Thus, the length of the first loop is an upper bound for the number of occurrences of any other loop between h_1 and h_2:

Suppose that there are at least l_0 loops of length l between positions h_1 and h_2, for some l. They are not necessarily adjacent—they can start in different memory states $k_1, k_2, \cdots, k_{l_0}$—but they are all of equal length l (see Fig. 5). Let d_i be the length of the input tape segment traversed by this computation path between k_i and k_{i+1}, for each $i = 0, \cdots, l_0 - 1$. (d_{l_0} denotes the length of the last segment between the end of the last loop and position h_2.)

But then, by the Position Independence Lemma, the machine A can get from k_i to k_{i+1} while moving its input head d_i positions to the right at any place of the input tape, provided that both the initial and final positions of this segment lie at least M^2+1 positions away from either margin.

Therefore, we can remove the l_0 loops of length l from the computation and shift all the segments of lengths $d_0, d_1, \cdots, d_{l_0-1}$ to the right so that the machine A, starting from memory state k_0 at position $h_1' = h_1 + l_0 l$, will be able to reach memory state q_2 at the right endmarker (see Fig. 6).

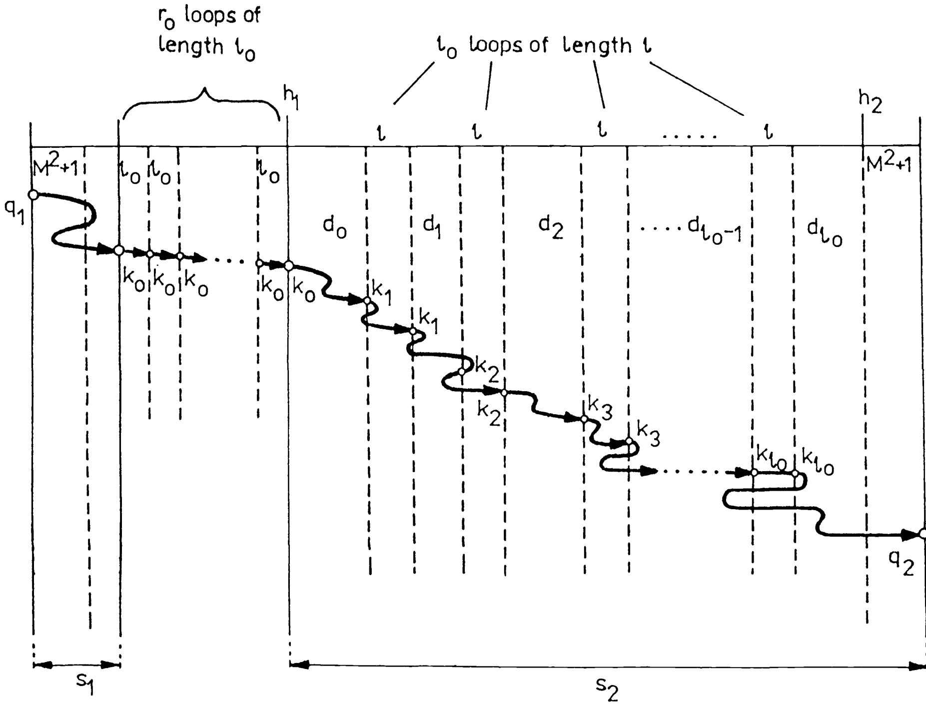


FIG. 5

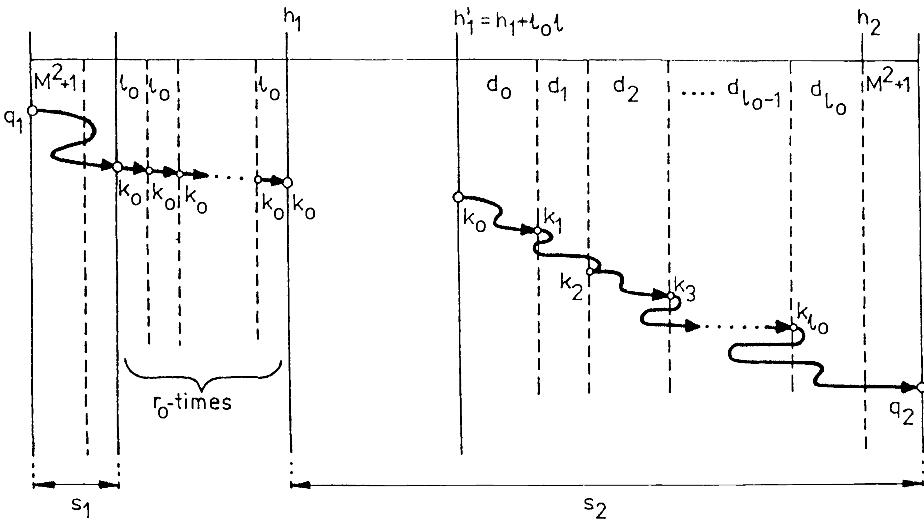


FIG. 6

To again get a valid computation path from q_1 to q_2, it is sufficient to show that the machine is able, starting from k_0 at position h_1, to get to the same memory state k_0 at position $h_1' = h_1 + l_0 l$. This is very easy, since, by the Position Independence Lemma, we can repeat the loop of length l_0 beginning in memory state k_0 l times (see Fig. 7).

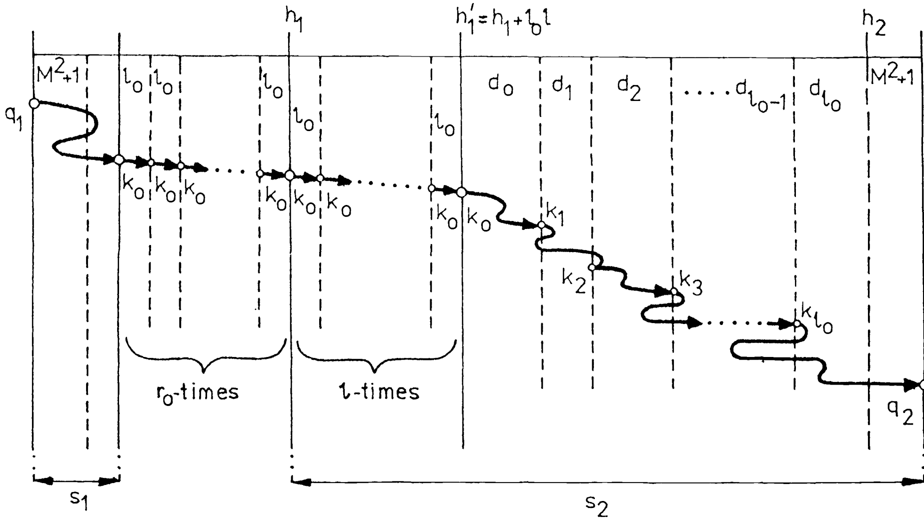


FIG. 7

Thus, we have found a computation path starting in q_1 and ending in q_2 such that A, after traversing s_1 positions, gets into a loop of length l_0 which is repeated $r_0' = r_0 + l$ times, and then traverses the rest of the input of length $s_2' = s_2 - l_0 l < s_2$. But this is a contradiction, since we assumed that s_2 was minimal.

Therefore, in computation path with minimal s_2, no loop of any length l can occur l_0 times between positions $h_1 = s_1 + r_0 l_0$ and $h_2 = \check{n} + \check{n}! - (M^2 + 1)$, since we can always replace l_0 loops of length l by l loops of length l_0.

Now, divide the input tape between positions h_1 and h_2 into adjacent segments of length $M+1$ (the last one can be of length $m \leqq M+1$) (see Fig. 8).

The machine must, in each segment of length $M+1$, get into a loop not longer than M. (To avoid any ambiguity, take the first, i.e., the leftmost loop in each segment.) Let r_l be the number of segments with the length of the leftmost loop exactly equal to l, for each $l = 1, \cdots, M$. (If for some l no segment has a leftmost loop of length l, then $r_l = 0$.) Then s_2 can be expressed in the form

$$s_2 = \sum_{l=1}^{M} r_l(M+1) + m + (M^2+1) \leqq M^2(M+1) + (M+1) + (M^2+1),$$

since $m \leqq M+1$, and $r_l \leqq l_0 \leqq M$ for each l, because no loop of any length can occur more than l_0 times between h_1 and h_2. But then, using (4), we have that $s_2 \leqq M^4$, and

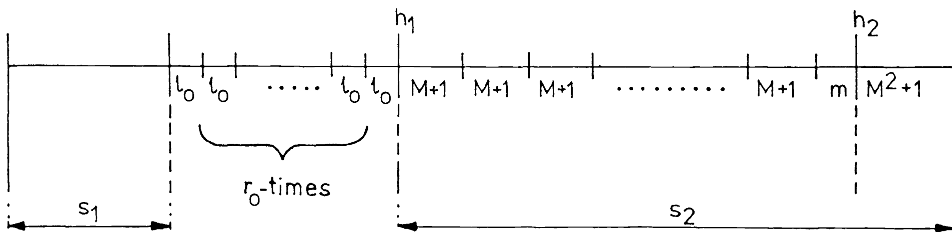


FIG. 8

thus, by combining this with (6), we obtain

$$1 \leqq l_0 \leqq M,$$
$$M^2+1 \leqq s_1 \leqq M^4,$$
$$M^2+1 \leqq s_2 \leqq M^4,$$

which completes the proof of the theorem. □

The next theorem asserts that traversals of A on the inputs $1^{\check{n}}$ and $1^{\check{n}+\check{n}!}$ are exactly the same, unless the space used exceeds $\check{s}$.

THEOREM 2 (Traverse). *The following holds for all memory states q_1, q_2 such that $/q_1/ \leqq /q_2/ \leqq \check{s}$: The machine A can get from the memory state q_1 to q_2 by traversing the whole input $1^{\check{n}}$ (the endmarkers are visited only in q_1 and q_2), if and only if A can get from q_1 to q_2 also by traversing $1^{\check{n}+\check{n}!}$.*

Proof. (A) $(\check{n} \to \check{n}+\check{n}!)$.

By (4), $M+1<\check{n}$, and therefore, if A traverses the whole input $1^{\check{n}}$ then it must get into a loop of length $l \leqq M < \check{n}$. But then A can also traverse the input $1^{\check{n}+\check{n}!}$, since this loop of length l can be iterated $\prod_{i=1, i\neq l}^{\check{n}} i$ more times.

(B) $(\check{n}+\check{n}! \to \check{n})$.

Suppose that A gets from q_1 to q_2 traversing the input $1^{\check{n}+\check{n}!}$. Then, by the Dominant Loop Theorem, A can also traverse this input by a computation path such that there exist s_1, l_0, r_0, s_2 satisfying (5) such that A, having traversed s_1 positions, gets into a loop of length l_0 which is repeated r_0 times, and then traverses the rest of the input of length s_2. Note that $\check{n}+\check{n}! = s_1 + r_0 l_0 + s_2$ and therefore

$$\check{n}+\check{n}! = s_1 + r_0 l_0 + s_2 \leqq 2M^4 + r_0 l_0 \leqq \tfrac{2}{6} M^5 + r_0 l_0 \leqq \check{n} + r_0 l_0,$$

by (4), and hence

$$r_0 l_0 \geqq \check{n}! = l_0 \cdot \prod_{\substack{i=1 \\ i \neq l_0}}^{\check{n}} i = l_0 \cdot F,$$

since $l_0 \leqq M < \check{n}$. Thus, $r_0 \geqq F$, i.e., there is a loop of length l_0, which is repeated at least F times. The first F iterations of this loop traverse exactly $l_0 \cdot F = \check{n}!$ positions to the right, beginning and ending in the same memory state k_0 (see Fig. 9).

Moreover, both the initial and final positions of this composite loop of length $\check{n}!$ beginning in k_0 lie at least M^2+1 positions away from either margin, since $s_1 \geqq M^2+1$, and $s_2 \geqq M^2+1$. From this we see that this $\check{n}!$-loop can be cut out of the computation

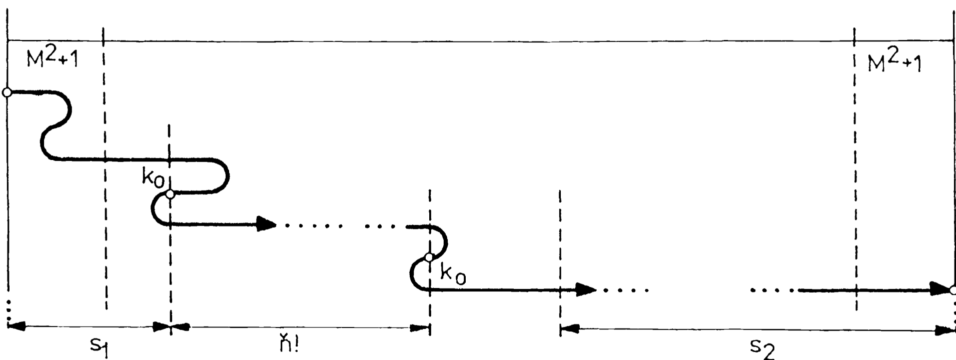


FIG. 9

path so that we obtain a valid computation traversing the input $1^{\check{n}}$, without having the head hit either margin. □

Now we can show that the behavior of A on the margins of $1^{\check{n}}$ and $1^{\check{n}+\check{n}!}$ is exactly the same, i.e., the set of memory states not bigger than $\check{s}$ reachable on the margins is the same.

LEMMA 5 (Inshore Fishing). *The following holds for each memory state k such that* $/k/ \leqq \check{s}$:

The machine A, starting from the initial memory state k_I *with the input head at the left endmarker, can get to the memory state k at the left (right) endmarker of the input* $1^{\check{n}}$ *if and only if A can also get to k on the corresponding endmarker of* $1^{\check{n}+\check{n}!}$.

Proof. (A) From the Traverse Theorem it follows that the machine A can get from q_1 to q_2 traversing the whole input $1^{\check{n}}$ from left to right if and only if it is also possible for the input $1^{\check{n}+\check{n}!}$. The same also holds, due to symmetry reasons, for traversals from right to left.

(B) The same can be shown for U-turns at the endmarkers of $1^{\check{n}}$ and $1^{\check{n}+\check{n}!}$, since, using the U-Turn Lemma, we can replace each U-turn by an equivalent U-turn not moving the input head farther than $M^2 < \check{n}$ positions away from an endmarker.

(C) The rest of the argument is a straightforward induction on the number of times the head visits the endmarkers. □

The above result can be extended for all reachable memory states not bigger than $\check{s}$.

LEMMA 6 (Deep-sea Fishing). *The following holds for each memory state k such that* $/k/ \leqq \check{s}$: *The memory state k is reachable on the input* $1^{\check{n}}$, *from the initial memory state* k_I *at the left endmarker, if and only if it is also reachable on the input* $1^{\check{n}+\check{n}!}$ *(but not necessarily at the same position).*

Proof. We shall show that all memory states not bigger than $\check{s}$ reachable on input $1^{\check{n}+\check{n}!}$ are also reachable on input $1^{\check{n}}$. (The converse is also true, by a very similar argument.)

Let q_1 be the last memory state of the computation path from k_I to k such that the input head was at an endmarker of the input $1^{\check{n}+\check{n}!}$. By the Inshore Fishing Lemma, q_1 is also reachable on the input $1^{\check{n}}$. There is no loss of generality in assuming that q_1 was at the left endmarker. Now, we have that A, starting from q_1 at the left endmarker of the input $1^{\check{n}+\check{n}!}$, can get to k at some position i (see Fig. 10).

But then A can also reach k from q_1 at a position $i' \leqq (M^2+1)+(M+1)$, for, if $i > (M^2+1)+(M+1)$, then we can find a loop between positions M^2+1 and $(M^2+1)+(M+1)$. This loop can be, by the Position Independence Lemma, removed from the computation path. This process can be repeated until we obtain a position $i' \leqq (M^2+1)+(M+1)$. But $(M^2+1)+(M+1)+(M^2+1) < \check{n}$, by (4), and therefore k is reachable from q_1 on the input $1^{\check{n}}$. □

We can now state and prove the main theorem.

THEOREM 3 (Constructibility). *If* $s(n)$ *is nondeterministically fully space constructible and* $\sup_{n\to\infty} s(n)/\log(n) = 0$, *then there exists* $\check{n}$ *such that for each* $n \geqq \check{n}$

$$s(n) = s(n+n!) = s(n+2n!) = s(n+3n!) = \cdots.$$

Therefore, there is no function $s(n)$ *such that*

(1) $s(n)$ *is unbounded (i.e., for each h there exists n with* $s(n) \geqq h$),

(2) $s(n)$ *is monotone increasing (i.e.,* $s(n) \leqq s(n+1)$, *for each n*),

(3) $\sup_{n\to\infty} s(n)/\log(n) = 0$,

(4) $s(n)$ *is fully space constructible by a nondeterministic Turing machine.*

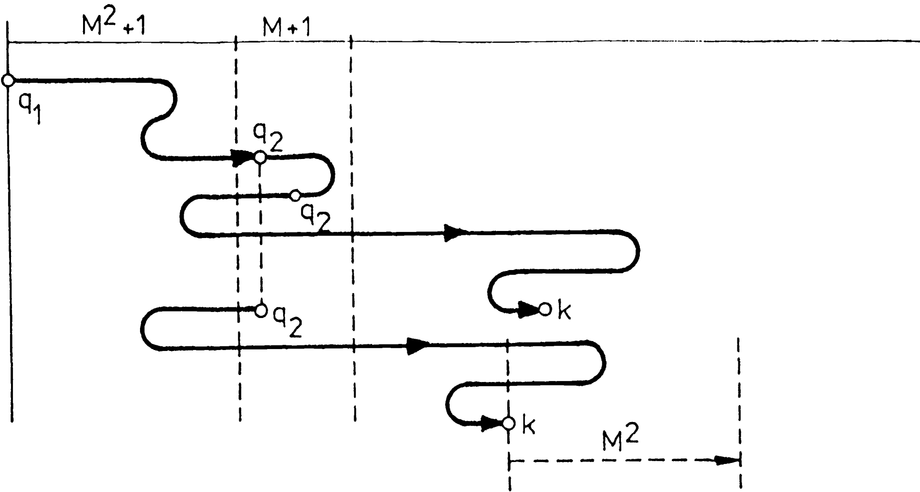


FIG. 10

Proof. (A) For each nondeterministically space-constructible function with $\sup_{n\to\infty} s(n)/\log(n) = 0$, we have, by the Deep-Sea Fishing Lemma, that the set of reachable memory states not bigger than $\check{s} = s(\check{n})$ is exactly the same for the inputs $1^{\check{n}}$ and $1^{\check{n}+\check{n}!}$. But all of the arguments above also hold for $1^{\check{n}}$ and $1^{\check{n}+u\check{n}!}$, for each $u \geqq 1$. We only have to replace $\check{n}!$ everywhere by $u\check{n}!$. Thus, $s(\check{n}+u\check{n}!) \geqq s(\check{n})$ for each $u \geqq 1$.

Now, suppose that $s(\check{n}+u\check{n}!) > s(\check{n})$ for some $u \geqq 1$. Since A must use exactly $s(\check{n}+u\check{n}!)$ space for $1^{\check{n}+u\check{n}!}$ on at least one computation path, there must be a memory state k_1 of size $\check{s} = s(\check{n})$ reachable on $1^{\check{n}+u\check{n}!}$ such that A, executing a single computation step, enters some memory state k_2 of size $s(\check{n})+1$. But then A can enter k_2 of size $s(\check{n})+1$ on $1^{\check{n}}$ (by a single step, from k_1), which is a contradiction, because A is $s(n)$ space bounded.

Therefore, $s(\check{n}) = s(\check{n}+u\check{n}!)$, for each $u \geqq 1$.

(B) Let $s(n)$ also be monotone increasing. Then, by (A), we have that $s(n) = s(\check{n})$ for each $n \geqq \check{n}$. Hence, $s(n)$ cannot be unbounded.

(C) We can easily extend (A) for all $n \geqq \check{n}$ (i.e., $s(n) = s(n+un!)$, for each $n \geqq \check{n}$ and each $u \geqq 1$), since all arguments throughout this section clearly hold if $\check{n}$ is replaced by $n > \check{n}$. □

4. Some consequences. In this section we discuss, without formal proofs, some properties of functions constructed by nondeterministic Turing machines working in sublogarithmic space. Then we shall present a characterization theorem for languages recognizable by such machines, together with lower bounds for some languages and a few separation results.

Remark 1. Functions $\log\log(n)$ and $\sqrt{\log(n)}$ are not nondeterministically fully space constructible, since every monotone increasing function below $\log(n)$ is either bounded by a constant, or not constructible by nondeterministic machines.

Remark 2. If $s(n)$, with $\sup_{n\to\infty} s(n)/\log(n) = 0$, is nondeterministically space constructible, then $\inf_{n\to\infty} s(n) =$ constant. (This follows from the fact that we can find $\check{n}$ such that $s(\check{n}) = s(\check{n}+u\check{n}!)$, for each $u \geqq 1$.)

Therefore, the lowest nondeterministically space-constructible upper bound for $\log\log(n)$ (or any unbounded monotone increasing function below $\log(n)$) is $\log(n)$.

Remark 3. None of the four conditions in the Constructibility Theorem can be omitted, since we can find functions satisfying each triple of these conditions:

2, 3, 4 (bounded): $s(n) = \text{constant}$.

1, 3, 4 (not monotone increasing): $s(n) = \log_2(fpr(n))$, where $fpr(n)$ = the first prime not dividing n.

$s(n)$ is space constructible within space $s(n)$, and $\sup_{n\to\infty} s(n)/\log(n) = 0$, because the language $L = \{1^n\colon (\exists k)\ n = p_1 p_2 \cdots p_k p_{k+2} t$ such that p_{k+1} does not divide $t\}$ is in SPACE $(\log\log(n))$. (Here p_i denotes the ith prime. For a more detailed proof, see [3].)

1, 2, 4 ($\sup_{n\to\infty} s(n)/\log(n) > 0$): $s(n) = \log(n)$.

1, 2, 3 (not fully space constructible nondeterministically): $s(n) = \sqrt{\log(n)}$.

COROLLARY 1. *For each unbounded function $s(n)$ below* $\log(n)$, *the function* $f(n) = \max\{s(1), \cdots, s(n)\}$ *is not fully space constructible by any nondeterministic Turing machine.*

(It follows from the fact that if $s(n)$ is unbounded with $\sup_{n\to\infty} s(n)/\log(n) = 0$, then so is $f(n)$. Moreover, $f(n)$ is always monotone increasing.)

This separates nondeterministic sublogarithmic space-bounded computations from the corresponding low-level nondeterministic space-bounded computations for computational models in which it is possible to construct the maximum of the first n values of sublogarithmic $s(n)$ within sublogarithmic space (for example, Turing machines which have an inkdot or pebble). We will return to these models later.

The next theorem characterizes languages recognizable within sublogarithmic space, and might be considered a sublogarithmic counterpart of the "Pumping Lemma" for context-free languages.

THEOREM 4 (Explosion). *For each $s(n)$ space-bounded nondeterministic Turing machine A recognizing a language $L \subseteq \{0, 1\}^*$ with* $\sup_{n\to\infty} s(n)/\log(n) = 0$, *and each $i \geqq 1$, there exists $\check{n} \geqq 2$ such that, for each $n \geqq \check{n}$, and each $m \geqq \sqrt[i]{n}$, $u1^m v$ of length n is in L if and only if $u1^{m+km!}v$ is in L, for each $k \geqq 1$.*

The above theorem, roughly speaking, asserts that, for sufficiently large n, there is a "critical length" $m = \sqrt[i]{n}$ such that if an accepted (rejected) word of length n contains a segment of at least m consecutive identical symbols, then this segment "explodes," i.e., the machine must also accept (reject) infinitely many other words containing longer and longer segments of these symbols.

The proof is analogous to the proof of our main result. The theorem is valid even if the constant i is replaced by a function $i(n)$ such that $\sup_{n\to\infty} i(n)s(n)/\log(n) = 0$. The critical explosive length for inputs of length n is then $m(n) = n^{1/i(n)}$. Therefore, there must be at least one critical explosive segment in each string with small information content, e.g., in a string consisting of at most $n^{1-1/i(n)}$ segments of consecutive identical symbols.

This theorem can be used for proving that some languages are not recognizable within sublogarithmic space.

COROLLARY 2. *Languages*

$$L_1 = \{a^n b^n;\ n \geqq 1\},$$

$$L_2 = \{a^n b^m;\ n \neq m\},$$

$$L_3 = \{w \in \{a, b\}^*;\ N_a(w) = N_b(w)\},$$

$$L_4 = \{w \in \{a, b\}^*;\ N_a(w) \neq N_b(w)\},$$

are not in NSPACE $(s(n))$, *for any* $s(n)$ *below* $\log(n)$. ($N_a(w)$ *denotes here the number of a's in the word w.*)

This reveals that the method of deterministic counting is the best one in some cases, and no nondeterministic trick can be used to reduce the space used.

When talking about sublogarithmic space, we must take the utmost care not to confuse different definitions of $s(n)$ space-bounded Turing machines in the literature (compare, for example, [2] with [4]). We now review the most typical variants.

DEFINITION 6. A nondeterministic Turing machine A is

(a) *weakly* $s(n)$ *bounded* if, for each w in $L(A)$ of length n, there exists *at least one* accepting computation path using no more than $s(n)$ space on the worktape,

(b) $s(n)$ *bounded* (or *strongly* $s(n)$ *bounded*) if, for each w of length n, *no* computation path uses more than $s(n)$ space,

(c) *super-strongly* $s(n)$ *bounded* if, for each w of length *at most n*, all computation paths are $s(n)$ bounded.

Let NWEAKSPACE $(s(n))$, NSPACE $(s(n))$, and NSSTRONGSPACE $(s(n))$ denote, respectively, the classes of languages recognizable by weakly, strongly, and superstrongly $s(n)$ space-bounded nondeterministic Turing machines, and let WEAKSPACE, SPACE, and SSTRONGSPACE be their deterministic variants.

(For example, deterministic weakly space-bounded Turing machines may use an arbitrary space for inputs which are not accepted.)

Clearly, NSSTRONGSPACE $(s(n)) \subseteq$ NSPACE $(s(n)) \subseteq$ NWEAKSPACE $(s(n))$ for each $s(n)$. Moreover, NSPACE $(s(n))$ = NWEAKSPACE $(s(n))$ for each nondeterministically space-constructible $s(n)$ and NSSTRONGSPACE $(s(n))$ = NSPACE $(s(n))$ for monotone increasing $s(n)$. But for each unbounded function $s(n)$ below $\log(n)$ we have that $s(n)$ is either not monotone increasing, or not nondeterministically fully space constructible.

We shall now turn to some modified computational models which were described in [3].

DEFINITION 7. (a) *A 1-inkdot Turing machine* is a usual Turing machine with the additional power of marking one tape square on the input with an inkdot. (This tape square is marked once and for all, no erasing.)

(b) *A pebble machine* is a Turing machine with the additional power of using one pebble, which can be placed on and then removed from the input tape arbitrarily many times.

(c) *A demon* $s(n)$ *machine* is a Turing machine with a two-way read-only input tape, and a two-way read-write worktape enclosed in endmarkers $s(n)$ positions apart for inputs of length n.

Let NSPACE* $(s(n))$, NPEBBLESPACE $(s(n))$, and NDEMONSPACE $(s(n))$ denote the classes of languages recognizable by 1-inkdot, pebble, and demon nondeterministic $s(n)$ space-bounded Turing machines, respectively. (We will also use their deterministic variants.)

Note that nondeterministic 1-inkdot Turing machines can compute, within space $\max\{s(1), \cdots, s(n)\}$, the value of $\max\{s(1), \cdots, s(n)\}$, for any nondeterministically space-constructible $s(n)$, be it below $\log(n)$ or not. (The machine will nondeterministically guess i between 1 and n with maximal $s(i)$, mark this position on the input tape with a dot, and then, using as its effective input the string between the left endmarker and the dot, compute the value of $s(i)$.)

The same holds for deterministic pebble machines and deterministically space-constructible functions, for the machine can use its removable pebble to compute, one by one, each of the values $s(1), \cdots, s(n)$.

Now, using the fact that $\{a^n b^n;\ n \geqq 1\}$ is in DEMONSPACE $(\log\log(n))$ and PEBBLESPACE $(\log\log(n))$, and that $\{a^n b^m;\ n \neq m\}$ is in NSPACE* $(\log\log(n))$ and WEAKSPACE $(\log\log(n))$, (for proofs, see [3] and [12]), and by Corollary 2, we have the following corollary.

COROLLARY 3. *For each* $s(n)$ *between* $\log\log(n)$ *and* $\log(n)$:

(a) SPACE $(s(n))$ = SPACE* $(s(n))$,
NSPACE $(s(n)) \subsetneq$ NSPACE* $(s(n))$,

(b) NSPACE $(s(n)) \subsetneq$ NDEMONSPACE $(s(n))$,
DEMONSPACE $(\log\log(n))$ − NSPACE $(s(n)) \neq \emptyset$,

(c) NSPACE $(s(n)) \subsetneq$ NWEAKSPACE $(s(n))$,
WEAKSPACE $(\log\log(n))$ − NSPACE $(s(n)) \neq \emptyset$,

(d) NSPACE $(s(n)) \subsetneq$ NPEBBLESPACE $(s(n))$,
PEBBLESPACE $(\log\log(n))$ − NSPACE $(s(n)) \neq \emptyset$.

Thus, even deterministic weakly space-bounded machines can be, in some cases, better than strongly bounded nondeterministic ones. The use of a single inkdot also increases the power of sublogarithmic nondeterministic machines. (The situation is quite different in deterministic case [3], or for $s(n)$ above $\log(n)$.)

5. Conclusion. We have shown that no unbounded monotone increasing function with $\sup_{n\to\infty} s(n)/\log(n) = 0$ is fully space-constructible by nondeterministic machines. This indicates that the sublogarithmic nondeterministic Turing machines are not very powerful and that the separation of nondeterministic space from deterministic space is harder than one might expect, even in a very low-level complexity range. Consequently, this main separation problem remains open.

On the other hand, it has some other consequences. We would like to conclude this paper with one of them, a tally version of the famous Savitch and Immerman–Szelepcsényi theorems. It can be shown that, for tally sets (i.e., $L \subseteq 1^*$), these two theorems are valid for each $s(n)$, be it space constructible or not, below $\log(n)$ or not. Some extension for more general sets can be made, but we do not know the answer in the most general case, i.e., whether these two theorems can also be proved for functions below $\log(n)$, i.e., for each $s(n)$, and each $L \subseteq \{0, 1\}^*$. Related results for the deterministic case can be found in [9].

Acknowledgment. The author thanks Branislav Rovan, Dana Pardubská, and the referees for several helpful suggestions and remarks concerning this work.

REFERENCES

[1] A. R. FREEDMAN AND R. E. LADNER, *Space bounds for processing counterless inputs*, J. Comput. System Sci., 11 (1975), pp. 118–128.

[2] M. A. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.

[3] J. HARTMANIS AND D. RANJAN, *Space bounded computations: Review and new separation results*, in Proc. Symp. on Mathematical Foundations of Computer Science, 1989, Lecture Notes in Computer Science 379, Springer-Verlag, Berlin, 1989, pp. 49–66.

[4] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[5] N. IMMERMAN, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17 (1988), pp. 935–938.

[6] P. M. LEWIS II, R. E. STEARNS, AND J. HARTMANIS, *Memory bounds for recognition of context-free and context-sensitive languages*, IEEE Conference Record on Switching Circuit Theory and Logical Design, Ann Arbor, MI, 1965, pp. 191–202.

[7] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[8] J. SEIFERAS, *A note on notions of tape constructibility*, Tech. Report CSD-TR 187, Pennsylvania State University, University Park, PA, 1976.

[9] M. SIPSER, *Halting space-bounded computations*, Theoret. Comput. Sci., 10 (1980), pp. 335-338.

[10] R. E. STEARNS, J. HARTMANIS, AND P. M. LEWIS II, *Hierarchies of memory limited computations*, IEEE Conference Record on Switching Circuit Theory and Logical Design, 1965, pp. 179-190.

[11] R. SZELEPCSÉNYI, *The method of forced enumeration for nondeterministic automata*, Acta Informatica, 26 (1988), pp. 279-284.

[12] A. SZEPIETOWSKI, *Some remarks on the alternating hierarchy and closure under complement for sublogarithmic space*, Inform. Process. Lett., 33 (1989), pp. 73-78.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 499-505, June 1991

© 1991 Society for Industrial and Applied Mathematics
007

A $4n$ LOWER BOUND ON THE COMBINATIONAL COMPLEXITY OF CERTAIN SYMMETRIC BOOLEAN FUNCTIONS OVER THE BASIS OF UNATE DYADIC BOOLEAN FUNCTIONS*

URI ZWICK†

Abstract. A simple, and easy-to-check, property of a symmetric boolean function is shown to imply a $4n - O(1)$ lower bound on the circuit complexity of the function over $U_2 = B_2 - \{\oplus, \equiv\}$, the basis of unate dyadic boolean functions. Among the functions to which this lower bound applies are the modular functions $\text{MOD}_k(n)$ for any fixed $k \geqq 3$ ($\text{MOD}_k(n)$ is the function which returns 1 if and only if $(\sum x_i) \bmod k = 0$). Finally, a $5n$ upper bound is obtained on the circuit complexity over U_2 of the function $\text{MOD}_4(n)$.

Key words. combinational complexity, boolean functions, lower bounds

AMS(MOS) subject classification. 68Q15

1. Introduction. In 1949, Shannon [Sh] showed that the circuit complexity of almost all boolean functions is exponential. However, attempts to obtain concrete lower bounds for functions in NP (see [KM], [HHS], [Sc-1], [Sc-2], [P], [St], [B]) yielded only linear results. The best lower bound of this kind known today over B_2, the full binary basis, is a $3n$ lower bound obtained by Blum [B]. Surveys of these results may be found in [BS], [D], [W].

In this note, we show that a $4n - O(1)$ lower bound may be proved if the linear functions XOR and its complement are removed from the basis. The best previous lower bound over this basis, denoted by U_2, was a $3n$ lower bound on the circuit complexity of the function $\text{MOD}_2(n)$ obtained by Schnorr [Sc-1]. The result presented here is, in a sense, a generalization of Schnorr's result. Another result which is close in spirit to the result presented here is the $2.5n$ lower bound (over B_2) obtained by Stockmeyer [St]. Some of the ideas in this work were inspired by the work of Lai and Muroga [LM].

For additional lower bounds over the bases $\{|\}$, $\{\vee, \neg\}$, $\{\vee, \wedge, \neg\}$, see [So] and [Re] (the symbol $|$ denotes the NAND operation).

In proving the $4n - O(1)$ lower bound we use a simple variation of the elimination method. It is very unlikely that this method will enable us to produce significantly better (for example, nonlinear) lower bounds. In order to achieve such an improvement, a major breakthrough, like the one recently obtained in the theory of monotone circuits (see [Ra], [A], [AB]), is probably needed.

2. Preliminaries. A *circuit* (over U_2) is a directed acyclic graph whose nodes have indegree 0 or 2. Nodes with indegree 0 are called *inputs* and they are labeled by variables or constants. Nodes with indegree 2 are called *gates* and they are labeled by functions from U_2. Note that we may assume, without loss of generality, that all the gates are labeled by the eight nondegenerate U_2-functions $(x^a \wedge y^b)^c$, where $x^a = x \oplus a$.

Each gate in a circuit computes a function by applying the function labeling it to the functions computed by the nodes feeding it. Since we are interested in this note in the computation of scalar functions, we consider only those circuits in which just

* Received by the editors November 16, 1988; accepted for publication (in revised form) April 18, 1989. This paper formed part of a Ph.D. thesis written by the author in Tel-Aviv University under the supervision of Professor Noga Alon.

† Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Present address, The Mathematical Institute, University of Warwick, Coventry CV4 7AL, United Kingdom.

one gate has outdegree 0. This gate is called the *output gate* of the circuit, and the function computed by it is defined to be the function computed by the circuit.

The *size* of a circuit β, denoted by $C(\beta)$, is the number of gates contained in it. The *circuit complexity* of a function f, denoted by $C_{U_2}(f)$ or simply by $C(f)$, is the minimal size of a circuit computing f.

If β is a circuit and A is a gate in β, we denote by $d_\beta(A)$ the outdegree of A, and by $\text{res}_\beta(A)$ the function computed at A (the subscript β is omitted when no confusion arises). If x is a variable, we denote by $d_\beta(x)$ the sum of the outdegrees of the input nodes labeled by x. Actually, we can assume that each variable labels only one input node and then $d_\beta(x)$ is simply the outdegree of the input node labeled by x.

Finally, we denote by $d_1(\beta)$ the number of variables whose outdegree in β is exactly 1. The numbers $d_1(\beta)$ play a central role in the proof of the $4n$ lower bound.

If a gate A in a circuit β is fed by the node B and $\text{res}(B)$ is constant, then we can obtain a smaller circuit which computes f in the following way: If $\text{res}(A)$ is also constant, then we simply remove the incoming edges of A from the circuit and turn A into a constant input node. Otherwise, $\text{res}(A)=\text{res}(C)^a$, where C is the second node in the circuit feeding A and $a\in\{0,1\}$. In this case we remove the node A and all its incoming and outgoing edges from the circuit. The gates which were fed by A will now be fed directly by C. If $a=1$, a complement must be incorporated into these gates. (We assume here that A is not the output gate of the circuit.) This process can be carried on until a *simplified circuit* is obtained, i.e., a circuit with no constant input nodes and no gates with constant output.

In the next section, we encounter many situations in which we are given a circuit, some of whose gates are fed by constants. In each such case we explicitly identify a subset of these gates and remove them one by one, as explained in the previous paragraph.

If $g(x,y)\in U_2$, then there exists a constant $c\in\{0,1\}$ such that $g(c,y)$ is a constant. We say that the constant c *blocks* the function g. Note that this property does not hold for B_2 and this is why proving lower bounds over this base is a harder problem.

3. The lower bound. We begin by defining the set of functions for which our lower bound applies.

DEFINITION 3.1. The sets $S(n)$, $M_k(n)$, $N_l(n)$, $MN_{k,l}(n)$ are defined in the following way:

(1) $f\in S(n)$ if and only if $f(x_1,\cdots,x_n)$ depends only on $\sum_{i=1}^n x_i$. Functions belonging to $S(n)$ are called *symmetric functions.* If $f\in S(n)$ and $f(x_1,\cdots,x_n)=v_k$, where $k=\sum x_i$, we associate with f the binary word $v(f)=v_0v_1\cdots v_n$. The word $v(f)$ is called the *value vector* of f.

(2) $f\in M_k(n)$ if and only if $f\in S(n)$ and every restriction of f to a subset of k variables is not constant. It is easy to see that $f\in M_k(n)$ if and only if $v(f)$ does not have a constant subword (i.e., $000\cdots$ or $111\cdots$) of length $k+1$.

(3) $f\in N_l(n)$ if and only if $f\in S(n)$ and every restriction of f to a subset $\{y_1,\cdots,y_l\}$ of f's variables is not linear, i.e., not $y_1\oplus\cdots\oplus y_l$ or its complement. It is easy to see that $f\in N_l(n)$ if and only if $v(f)$ does not have an alternating subword (i.e., $0101\cdots$ or $1010\cdots$) of length $l+1$.

(4) Finally, we define: $MN_{k,l}(n)=M_k(n)\cap N_l(n)$. In other words, $f\in MN_{k,l}(n)$ if and only if $v(f)$ does not have a constant subword of length $k+1$ or an alternating subword of length $l+1$.

The structure of the sets $M_k(n)$, $N_l(n)$, $MN_{k,l}(n)$ for small k and l is very simple. It is easy to check that $M_1(n)$ contains only the two linear functions $x_1\oplus\cdots\oplus x_n\oplus c$

that have the value vectors $0101\cdots$ and $1010\cdots$ and that $N_1(n)$ contains only the two constant functions that have the value vectors $000\cdots$ and $111\cdots$. Consequently, the sets $MN_{k,1}(n)$ for $n \geqq k \geqq 1$ and $MN_{1,l}(n)$ for $n \geqq l \geqq 1$ are empty. The first nonempty set of the form $MN_{k,l}(n)$ is $MN_{2,2}(n)$. If $f \in MN_{2,2}$, then $v(f)$ does not contain the subwords 000, 111, 010, 101. The only words with this property are $00110011\cdots$, $0110011\cdots$, $11001100\cdots$, and $1001100\cdots$, and therefore:

$$MN_{2,2} = \left\{ \left\lfloor \frac{(\sum x_i + c) \bmod 4}{2} \right\rfloor : c = 0, 1, 2, 3 \right\}.$$

As a further example, we note that $\text{MOD}_k(n) \in MN_{k-1,3}(n)$ for $k > 2$.

If $f \in MN_{k,l}(n)$ for some $n \geqq k, l$, then $v(f)$ has a subword from the set {001, 110, 100, 011}. In particular, for every two variables $x, y \in \{x_1, \cdots, x_n\}$ the function f has a restriction of the form $(x^a \wedge y^a)^b$. It is also obvious that if $f \in MN_{k,l}(n)$ for some $n > k, l$, then every restriction of f obtained by fixing the value of one variable belongs to $MN_{k,l}(n-1)$.

We can now prove that if $f \in MN_{k,l}(n)$ and $n \geqq k+1, l$, then $C_{U_2}(f) \geqq 4(n-m)-1$, where $m = \max\{k+1, l\}$.

LEMMA 3.2. *Let β be a circuit which computes a function $f \in MN_{k,l}(n)$ for $n > k+1, l$. There exists a circuit δ which computes a function $f' \in MN_{k,l}(n-1)$ and which satisfies* $[C(\delta) - d_1(\delta)] \leqq [C(\beta) - d_1(\beta)] - 4$.

Proof. Let β be a circuit which computes a function $f \in MN_{k,l}(n)$, where $n > k+1, l$. If β is not simplified, then simplify it and denote the simplified circuit obtained by γ. It is easy to check that $[C(\gamma) - d_1(\gamma)] \leqq [C(\beta) - d_1(\beta)]$ (in every elementary simplification step described in the previous section, the size of the circuit decreased by 1 and d_1 could have increased by at most 1). In γ there exists a gate B, which is fed by two input variables. Let x, y be the variables feeding B. The outdegrees of x and y must be at least 2 (since otherwise we can assign a value to one of them and make the output independent of the other).

We say that the circuit γ is *degenerate* if it contains the situation shown in Fig. 3.1 (or the symmetrical situation with the roles of x and y switched). It is easy to transform γ into a nondegenerate circuit γ' which also computes f and which satisfies $C(\gamma') \leqq C(\gamma)$, $d_1(\gamma') = d_1(\gamma)$. This can be accomplished by either deleting A from the circuit, if its output does not depend on both x and y, or by replacing the edge $B \to A$ by an edge $y \to A$, and by adjusting the gate A if necessary, otherwise. Notice that in the latter case $\text{res}(A) = (x^a \wedge y^b)^c$ since in order to compute $(x \oplus y)^d$ at least three U_2-gates are needed. The only variables whose outdegree could have been changed by these actions are x and y. But the outdegrees of x and y were, and must remain, at least 2, and therefore $d_1(\gamma') = d_1(\gamma)$.

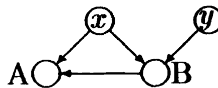


FIG. 3.1

We may therefore assume, without loss of generality, that the circuit γ is nondegenerate. We consider the following three cases.

Case 1. $d_\gamma(x) \geqq 3$ or $d_\gamma(y) \geqq 3$.

Assume without loss of generality that $d_\gamma(x) \geqq 3$. Let A, C be two additional gates fed by x, as shown in Fig. 3.2. Let D be a gate fed by B. Since γ is nondegenerate, $D \neq A, C$. We assign to x the constant c which blocks B and delete the gates A, B, C,

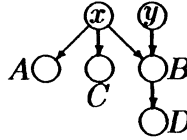


FIG. 3.2

D from β as explained in the previous section. Denote by δ the new circuit obtained. The circuit δ computes the function $f_{x:=c} \in MN_{k,l}(n-1)$. Note that if $d_\gamma(z)=1$, then z does not feed A, C, or D in γ (since otherwise choosing the right values for x and y would make the output of γ independent of z and this is a contradiction since $n > k+1$). Therefore, $d_\delta(z)$ remains 1 and thus we have $C(\delta)=C(\gamma)-4$ and $d_1(\delta) \geqq d_1(\gamma)$ as required.

If Case 1 does not hold, then $d_\gamma(x)=d_\gamma(y)=2$. Denote by A, B the gates fed by x and recall that B is also fed by y.

Case 2. $d_\gamma(B) \geqq 2$.

Denote by C, D two distinct gates fed by B (see Fig. 3.3). Since γ is nondegenerate C, $D \neq A$. We assign to x the constant c which blocks B. As in the previous case, we delete the gates A, B, C, D and we are left with a circuit δ which satisfies $C(\delta)=C(\gamma)-4$ and $d_1(\delta) \geqq d_1(\gamma)$, as required.

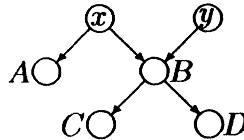


FIG. 3.3

The last case we have to consider is Case 3.

Case 3. $d_\gamma(B)=1$.

We break this case into two subcases.

Case 3.1. There exists an edge $y \to A$ in γ.

If $d_\gamma(A) \geqq 2$, then after switching the roles of x and y we are back in Case 2. We therefore assume that $d_\gamma(A)=d_\gamma(B)=1$. Denote by C the only gate fed by B, and by D the only gate fed by A. We claim that $C \neq D$, for otherwise the output of γ depends on x and y only through the gate $C=D$. If res (C) is constant or of one of the forms x^a, y^b, $(x^a \wedge y^b)^c$, then one of x or y may block the other, and this is clearly a contradiction. The only possibility left is that res $(C)=(x \oplus y)^d$ but this also leads to a contradiction, for in this case f cannot have a restriction to $\{x, y\}$ of the form $(x^a \wedge y^a)^b$.

The situation in this subcase is therefore as shown in Fig. 3.4. We assign to x the constant which blocks B, and delete the gates A, B, C from the circuit. Denote the resulting circuit by δ. Note that $d_\delta(y)=1$ since y feeds in δ only the gate D. Once again, if $d_\gamma(z)=1$, then also $d_\delta(z)=1$, and therefore $C(\delta) \leqq C(\gamma)-3$ and $d_1(\delta) \geqq d_1(\gamma)+1$, as required.

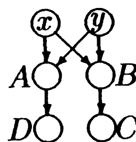


FIG. 3.4

Case 3.2. There is no edge $y \to A$ in γ.

Denote by C the unique gate fed by B. Denote by D the second gate fed by y. By the nondegeneracy of γ we obtain that $C \neq A$, D. We assume in this case that $D \neq A$, thus the situation is as shown in Fig. 3.5. As usual, we assign to x the constant which blocks B, and delete the gates A, B, C from the circuit. We can now repeat the arguments of the previous subcase.

In each one of the above cases we obtained a circuit which satisfies the conditions required and thus the proof is complete. □

THEOREM 3.3. *If* $f \in MN_{k,l}(n)$, *then* $C_{U_2}(f) \geqq 4(n-m)-1$, *where* $m = \max\{k+1, l\}$.

Proof. We prove by induction on n that if β is a circuit which computes $f \in MN_{k,l}(n)$, then $[C(\beta)-d_1(\beta)] \geqq 4(n-m)-1$. The basis of the induction for $n=m$ follows from the fact that $C(\beta) \geqq m-1$, $d_1(\beta) \leqq m$, and therefore $[C(\beta)-d_1(\beta)] \geqq -1$. The induction step follows immediately from Lemma 3.2. □

In fact, this theorem can be slightly improved to $C_{U_2}(f) \geqq 4(n-m)+(m-k)$ using the following lemma.

LEMMA 3.4. *If* $f \in M_k(n)$ *and* β *is a simplified circuit computing* f, *then* $d_1(\beta) < k$.

Proof. Suppose on the contrary that $d_\beta(x_1) = \cdots = d_\beta(x_k) = 1$. Denote by A_i the unique gate fed by x_i for $i \leqq k$. Since β is a simplified circuit, the second input of A_i is not constant. Denote by V_i the set of variables on which this second input depends. Since, by assigning appropriate values to the variables of V_i we can block x_i and thus obtain a constant restriction, we immediately get that $|V_i| \geqq n-k+1$. Thus each V_i contains at least one variable from the set $\{x_1, \cdots, x_k\}$. Denote one such variable by $x_{\pi(i)}$. For each $1 \leqq i \leqq k$ there exists a directed path in β from $x_{\pi(i)}$ to A_i and therefore also from $A_{\pi(i)}$ to A_i. But this is a contradiction since it implies the existence of a directed circuit in β. □

4. An upper bound. In this section, we present U_2-circuits of size $5n-7$ which compute the functions $\mathrm{MOD}_4(n)$ (for $n \geqq 3$). This shows that $4n - O(1) \leqq C_{U_2}(\mathrm{MOD}_4) \leqq 5n - O(1)$.

Using the same methods, it is easy to see that any symmetric boolean function $f(x_1, \cdots, x_n)$ which depends only on $(\sum x_i) \bmod 2^k$ can be computed using $(7-2^{3-k})n + O(2^k/k)$ U_2-gates or $(5-2^{3-k})n + O(2^k/k)$ B_2-gates. In particular, any symmetric boolean function can be computed using $7n+o(n)$ U_2-gates or $5n+o(n)$ B_2-gates.

The basic building block in our circuits is the binary full adder (FA) shown in Fig. 4.1(a). In Figs. 4.1(b) and 4.1(c) it is shown how an FA can be implemented using five B_2-gates or seven U_2-gates. It is easy to check that both implementations are optimal.

We now present the circuits for $\mathrm{MOD}_4(n)$. If $(s_{k-1}, \cdots, s_0)$ is the binary representation of $x_1 + \cdots + x_n$, then $\mathrm{MOD}_4(x_1, \cdots, x_n) = \mathrm{NOR}(s_1, s_0)$. The circuits compute s_1, s_0 using a tree of FAs.

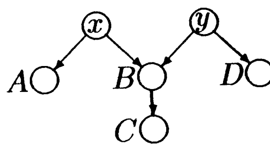


FIG. 3.5

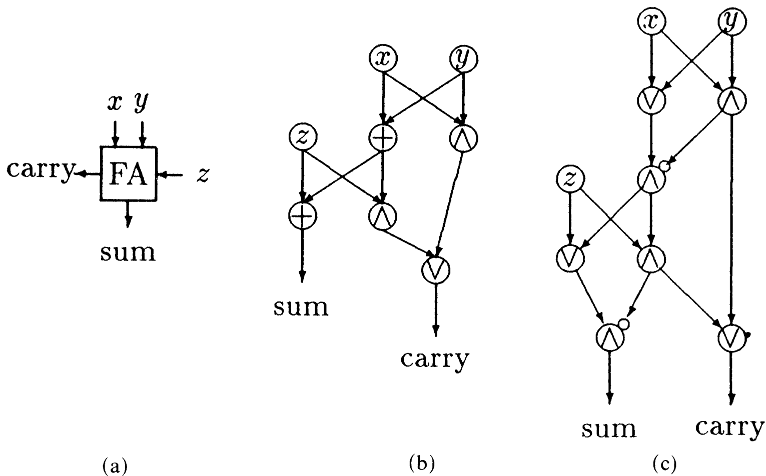


FIG. 4.1. *Implementation of full adders.*

Assume for simplicity that $n \geqq 3$ is odd. Construct a trenary tree of FAs in which every FA is fed by three inputs which are either input variables or sum outputs of previous FAs (this is possible since n is odd). The exact structure of the tree is immaterial. The number of FAs in the tree is $(n-1)/2$. The output of the root FA is s_0. The function s_1 is obtained by computing the XOR of all the carries produced by the $(n-1)/2$ FAs. This XOR can be computed using $3 \cdot ((n-1)/2) - 3$ U_2-gates. The value of the function $\text{MOD}_4(n)$ is now obtained using one additional NOR gate. The total size of the circuit is $7 \cdot ((n-1)/2) + (3 \cdot ((n-1)/2) - 3) + 1 = 5n - 7$.

The same construction yields B_2-circuits of size $3n-3$ for the functions $\text{MOD}_4(n)$. Stockmeyer [St] constructed more efficient circuits for $\text{MOD}_4(n)$ over B_2 and showed that $C_{B_2}(\text{MOD}_4(n)) = 2.5n - O(1)$.

Over U_2 there is still an unresolved gap between the $4n$ lower bound and the $5n$ upper bound presented for $\text{MOD}_4(n)$. We believe that the upper bound is closer to the truth and that the function $\text{MOD}_4(n)$ is the easiest function among the functions to which the lower bound presented in this note applies.

Acknowledgment. The author would like to thank Noga Alon for his help and supervision during the preparation of this work.

REFERENCES

[A] A. E. ANDREEV, *On a method for obtaining lower bounds for the complexity of individual monotone functions*, Dokl. Akad. Nauk. SSSR, 282 (1985), pp. 1033-1037. (In Russian.) English translation in Sov. Math. Dokl., 31 (1985), pp. 530-534.

[AB] N. ALON AND R. B. BOPPANA, *The monotone circuit complexity of boolean functions*, Combinatorica, 7 (1987) pp. 1-22.

[B] N. BLUM, *A Boolean function requiring 3n network size*, Theoret. Comput. Sci., 28 (1984), pp. 337-345.

[BS] R. B. BOPPANA AND M. SIPSER, *The complexity of finite functions*, in Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity, J. van Leeuwen, ed., Elsevier, Amsterdam, the Netherlands, pp. 757-800.

[D] P. E. DUNNE, *The complexity of Boolean Networks*, Academic Press, London, 1988.

[HHS] L. H. HARPER, W. N. HSIEH, AND J. E. SAVAGE, *A class of boolean functions with linear combinational complexity*, Theoret. Comput. Sci., (1975), pp. 161-183.

[KM] B. M. KLOSS AND V. A. MALYSHEV, *Bounds on complexity of some classes of functions*, Vestnik Moskov. Univ. Ser. Mat. Mekh., 4 (1965), pp. 44-51. (In Russian.)

[LM] H. C. LAI AND S. MUROGA, *Logic networks with a minimum number of* NOR (NAND) *gates for parity functions of n variables*, IEEE Trans. Comput., 36 (1987), pp. 157-166.

[P] W. J. PAUL, *A 2.5n-lower bound on the combinatorial complexity of boolean functions*, SIAM J. Comput., 6 (1977), pp. 427-443.

[Ra] A. A. RAZBOROV, *Lower bounds for the monotone complexity of some boolean functions*, Dokl. Akad. Nauk. SSSR, 281 (1985), pp. 791-801. (In Russian.) English translation in Sov. Math. Dokl., 31 (1985), pp. 354-357.

[Re] N. P. RED'KIN, *Proof of minimality of circuits consisting of functional elements*, Problemy Kibernet., 23 (1970), pp. 83-101. (In Russian.)

[Sc-1] C. P. SCHNORR, *Zwei Lineare untere Schranken für die Komplexität Boolescher Funcktionen*, Computing, 13 (1974), pp. 155-171.

[Sc-2] ———, *The combinational complexity of equivalence*, Theoret. Comput. Sci., 1 (1976), pp. 289-295.

[Sh] C. E. SHANNON, *The synthesis of two-terminal switching circuits*, Bell Systems Tech. J., 28 (1949), pp. 59-98.

[So] E. P. SOPRUNENKO, *Minimal realization of functions by circuits using functional elements*, Problemy Kibernet., 15 (1965), pp. 117-134.

[St] L. STOCKMEYER, *On the combinational complexity of certain symmetric boolean functions*, Math. Systems Theory, 10 (1977), pp. 323-336.

[W] I. WEGENER, *The Complexity of Boolean Functions*, Wiley-Teubner Ser. Comput. Sci., 1987.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 506-523, June 1991

© 1991 Society for Industrial and Applied Mathematics

008

NEAR-TESTABLE SETS*

JUDY GOLDSMITH†, LANE A. HEMACHANDRA‡, DEBORAH JOSEPH§, AND PAUL YOUNG¶

Abstract. In this paper a new property of sets, *near-testability*, is introduced. A set S is *near-testable* ($S \in NT$) if the membership relation for all immediate neighbors is polynomially computable; i.e., if the function $t(x) = \chi_S(x) + \chi_S(x-1) \pmod 2$ is polynomially computable. The near-testable sets form a subclass of the class $\oplus P$ (parity polynomial time), introduced by Papadimitriou and Zachos, and Goldschlager and Parberry. $\oplus P$ has a complete set $\oplus SAT$ that has recently been shown by Valiant and Vazirani to be hard for NP under randomized polynomial-time reductions. It is proved that there is a uniform polynomial one-one reduction that takes every set in $\oplus P$ to a near-testable set, and it is shown that the image of $\oplus SAT$ under this reduction (which we call $NTSAT$) is polynomially isomorphic to $\oplus SAT$. As corollaries it is shown that $NTSAT$ is complete for both NT and for $\oplus P$, that $NTSAT$ is hard for NP under randomized polynomial-time reductions, and that the existence of one-way functions implies the existence of sets that are near-testable but not polynomially decidable. It is then asked whether near-testability is preserved under p-isomorphisms. This leads to a generalization, NT^*, of NT similar to those introduced by Meyer and Paterson and by Ko for self-reducible sets. With this more general definition, NT^* is shown to be closed under polynomial-time isomorphisms while remaining a subclass of $\oplus P$. It is conjectured that it is a proper subclass. In fact it is shown that, relative to a random oracle, the containments $P \subseteq NT \subseteq NT^* \subseteq \oplus P$ are proper with probability one. It is also shown that, relative to a random oracle, with probability one NT and NT^* are incomparable with both NP and with $coNP$. Finally, the effects that the distribution and density of elements have on the complexity of near-testable sets are considered.

Key words. structural complexity theory, near-testability, one-way functions, parity polynomial time, self-reducibility

AMS(MOS) subject classification. 68Q15

C.R. subject classification. F.1.3

1. Introduction: Basic facts about near-testable sets.

Structural complexity theory is often concerned with the interrelationship between sets in a complexity class (e.g., *is set S complete for class C*?), and inclusion relationships between classes (e.g., *does* $P = NP$?). However, the internal properties of sets (e.g., *do all NP-complete sets have infinite polynomially decidable subsets*?) are also of interest. We are concerned here not just with the internal structure of sets, but more specifically with the *ordering structures* that exist *within a* set and with the effect that these orderings have on the time and space complexity of the set.

The study of sets and their associated orderings is not new. For example, the classes of *Turing self-reducible* sets, *p-selective* sets, and *p-cheatable* sets have each been widely studied, and the sets in each class have an internal partial ordering that is imposed by the nature of the "self-reducibility" the set exhibits. Turing self-reducible

* Received by the editors July 21, 1988; accepted for publication June 29, 1990. This work was supported in part by the National Science Foundation under grants DCR-8402375, DCR-8520597, CCR-8809174/8996198, and CCR-8957604, and also by a grant from AT&T Bell Laboratories. This paper extends results of a preliminary version presented at the Second Annual Structure in Complexity Theory Conference, Ithaca, New York, 1987 [GJY87a].

† Department of Computer Science, Boston University, Boston, Massachusetts 02215.

‡ Department of Computer Science, University of Rochester, Rochester, New York 14627.

§ Computer Sciences and Mathematics Departments, University of Wisconsin, 1210 West Dayton St., Madison, Wisconsin 53706.

¶ Department of Computer Science and Engineering FR-35, University of Washington, Seattle, Washington 98195.

sets have an internal well-founded ordering that ties the membership question for any element of the domain to the membership question for polynomially many "shorter" elements, and the p-selective sets impose on their domain a polynomially testable reflexive and transitive preorder.

This paper is part of a general investigation of how internal orderings affect the time and space complexities of sets. Reference [GJY87a] surveys results concerning various classes of "self-reducible" sets, outlining in a systematic way how ordering structures affect the time and space complexities of these sets, and calling for a "continuing systematic study of the relationship between... internal structure and the computational complexity of a set." References [GJY87b] and [GJY87c] go on to more fully investigate the time and space complexity of p-selective, p-cheatable, self-reducible, and word-decreasing-query self-reducible sets. Thus these investigations fall within a longstanding—yet newly reemergent—research stream, surveyed recently in [H90], that seeks to build a substantial collection of efficient operations that can be performed on sets whose *membership* problems seem to be complex.

More recent advances along this line include polynomial-time enumeration schemes for all NP-complete sets that have arisen by direct construction [HHSY] and polynomial-time minimal perfect static hash functions for all standard *NP*-complete sets [GHK90]. The present paper gives strong evidence that all sets in parity polynomial time are one-one reducible to sets whose membership problem can be partially tested in an efficient manner. Adapting and extending the techniques of this paper, [HH90] goes on to show that an even larger class of sets—$\oplus$ Opt P—can be one-one reduced to sets having slightly more general algorithms for partially testing membership.

This paper defines and studies the class of *near-testable* sets, a class of sets that have a particularly simple internal ordering structure. Like many other self-reducible sets, near-testable sets lie somewhere between P and $PSPACE$. Sets are near-testable if, in polynomial time, for any element x it is possible to fully relate the membership question for x to the membership question for x's immediate predecessor in the lexicographic ordering.[1] Because of this very simple structure, we hope that it will be possible to analyze near-testable sets in ways that are not possible for more complicated classes.[2]

This intuitive definition of near-testability is easily formalized as follows:

DEFINITION 1.1. A set S is near-testable if there is a polynomially computable function that, given $x>0$, decides whether exactly one of x and $x-1$ is in S. That is, the function

$$t(x)=\chi_S(x)+\chi_S(x-1) \pmod 2$$

is polynomially computable. We denote the class of near-testable sets by NT.

[1] Here and throughout this paper, we associate the integer n with the nth string in lexicographical order, so that we may speak interchangeably about strings and numbers with no distinctions between the two (see [MY78]). Thus, 0 will stand both for the number 0 and for the empty string.

[2] In [Ba87], Balcázar introduced the *word-decreasing-query self-reducible* sets. Although our original interest in near-testable sets was independent of Balcázar's work, in hindsight his definitions provide a nice motivation for studying near-testable sets. Balcázar said that a set S is *polynomial-time word-decreasing-query* (wdq) *self-reducible* if there is a polynomial-time deterministic oracle Turing machine M such that M^S decides membership in S and for each input x, all queries to S lexicographically precede x. Near-testable sets are easier to analyze than Balcázar's sets, since for near-testable sets the membership question for any given element can be reduced in polynomial time to the membership question for its *immediately* preceding neighbor.

Thus, with respect to our ordering on Σ^*, we can fully relate the membership questions for x and $x-1$, where $x-1$ denotes the lexicographic, or equivalently the numeric, predecessor of x. We easily see that if S is near-testable, then $\bar{S}$ is also, and that all near-testable sets are decidable in linear exponential time[3] and in polynomial space. In addition, it is easily seen that if S is near-testable and if either S or $\bar{S}$ is polynomially sparse, then S is in P. On the other hand, a straightforward diagonalization allows us to construct sets that are exponentially (or even just superpolynomially) decidable but not near-testable. We summarize these observations as follows.

OBSERVATION 1.2. (i) $P \subseteq NT = coNT \subseteq E \cap PSPACE$.

(ii) If $S \in NT$ and if either S or $\bar{S}$ is polynomially sparse, then $S \in P$.

(iii) There is a set in E that is not near-testable.

An important concept in the study of near-testable sets is the *boundary* of a set. Given a set S, the boundary of S will be defined by

$$\text{boundary}\,(S) = \{x\colon x-1 \text{ is defined and exactly one of } x \text{ and } x-1 \text{ is in } S\}.$$

Thus the boundary of a set S consists of the first element of every contiguous sequence of elements of S and of $\bar{S}$ with the exception of the first contiguous sequence.

We can picture a set S for which $0 \in \bar{S}$ as follows:

S (exclusive of points in its boundary) is represented by the thicker horizontal lines, $\bar{S}$ (exclusive of points in its boundary) by the thinner horizontal lines, and boundary (S) by lines of the form "|" and "$_{|}$". Note that boundary $(S) =$ boundary $(\bar{S})$.

OBSERVATION 1.3. S is near-testable if and only if boundary $(S) \in P$.

2. Relating near-testability and parity testing. Papadimitriou and Zachos [PZ83] and Goldschlager and Parberry [GP86] introduced the complexity class $\oplus P$, parity polynomial time, which has recently proved useful in the study of randomized polynomial-time reducibilities and in separating the polynomial hierarchy from *PSPACE*. We will see that the near-testable sets form a subclass of $\oplus P$, and perhaps surprisingly, every set in $\oplus P$ is one-one polynomial-time reducible ($\leqq_1^P$) to a near-testable set.

DEFINITION 2.1. A set S is in $\oplus P$ if there is a nondeterministic polynomial-time Turing machine M such that $x \in S$ if and only if the number of accepting paths in M's computation tree for input x is odd.

Papadimitriou and Zachos refer to $\oplus P$ as "a more moderate version" of Valiant's class $\#P$, which is the class of functions that count the number of accepting paths produced by nondeterministic polynomial-time Turing machines [Va79]. Clearly, if we could compute $\#P$, then $\oplus P$ would be no more difficult. Like $\#P$, $\oplus P$ is in *PSPACE* and is thought not to be contained in the polynomial hierarchy; Toda has recently shown that if $\oplus P$ is in the polynomial hierarchy then the hierarchy collapses [To88], and Regan and Royer have shown that $\oplus P$ strictly contains the polynomial hierarchy relative to a random oracle [RR90].

We next observe that the near-testable sets form a subclass of $\oplus P$.

THEOREM 2.2. $NT \subseteq \oplus P$.

Proof. Suppose that S is *near-testable*. Let us assume for the moment that $0 \in \bar{S}$. We will describe a nondeterministic *parity machine* M that recognizes S. On input x, M will guess a string lexicographically less than or equal to x, trying to guess an

[3] Throughout this paper we will use E to denote the class of sets recognizable in deterministic time $2^{O(n)}$ and *EXP* to denote the class of sets recognizable in time $2^{n^{O(1)}}$.

element of boundary (S) and verify its guess. Note that $x \in S$ *if and only if* the number of strings in boundary (S) that are less than or equal to x is odd *if and only if* there are an odd number of accepting computation paths for $M(x)$. If $0 \in S$, then we simply add one vacuous accepting path to each of M's calculations. □

The preceding proof shows the close connection between a set, the boundary of the set, and the *parity* of the boundary. In fact, for any set B if we let

$$\text{parity}_B(z) = \begin{cases} 1 & \text{if } |\{w \leqq z : w \in B\}| \text{ is odd,} \\ 0 & \text{otherwise,} \end{cases}$$

then, identifying a set with its characteristic function, it is easy to see for any set S that

$$(*) \qquad 0 \in \bar{S} \Rightarrow S = \text{parity}_{\text{boundary}(S)} \quad \text{and} \quad 0 \in S \Rightarrow \bar{S} = \text{parity}_{\text{boundary}(S)}.$$

Although it seems unlikely that all sets in $\oplus P$ are near-testable, we next show that these two classes are nevertheless very closely related.

THEOREM 2.3. *Every set in $\oplus P$ is $\leqq_1^P$-reducible to a near-testable set via a polynomially invertible reduction that has a polynomially decidable range.*[4]

Proof. We describe a polynomial-time procedure that, given a nondeterministic polynomial-time Turing machine M, constructs a near-testable set S so that $T \leqq_1^P S$, where T is the set accepted by M viewed as a $\oplus P$ machine—$T = \{x: M(x)$ has an odd number of accepting paths$\}$.

Let p be a polynomial bound on the running time of M. Without loss of generality we may make a number of assumptions about the polynomial bound p and the length of M's computations[5] on inputs of length n:

(i) That p is of the form $n^i + i$ for some i:
(ii) That all computations for an input of length n have length exactly $p(n)$; and
(iii) That no computation y is in 0^* or in 1^*.

To construct the set S we first construct a polynomially decidable set B. S is then completely defined by specifying that $0 \in \bar{S}$ and that boundary $(S) = B$. The basic idea is to let

$$B = \{\langle x, y\rangle : y \text{ is an accepting computation of } M(x)\},$$

where $\langle x, y\rangle = 0x_1 0x_2 \cdots 0x_n 1 y_1 y_2 \cdots y_k$, for $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_k$. This construction will yield only that T is polynomially truth-table reducible to S, so after seeing how the basic idea works we modify the definition of B to obtain $T \leqq_1^P S$.

When $|y| = p(|x|)$, we will call a pair (x, y) *relevant.* Note that (i) all $\langle x, y\rangle$'s precede all $\langle x+1, z\rangle$'s when (x, y) and $(x+1, z)$ are relevant pairs; (ii) for relevant pairs (x, y), $\langle x, y\rangle$ is computable in time polynomial in the length of x; and (iii) there is a decoding function for relevant pairs that is polynomially computable in the length of the string coding the pair.

Using this pairing function, B is obviously in P, and if we define S by requiring that $0 \in \bar{S}$ and boundary $(S) = B$, then by Observation 1.3, S is near-testable.

---

[4] Note that Theorems 2.2 and 2.3 do *not* guarantee that every *set* in $\oplus P$ is even polynomially many-one equivalent to a near-testable set. We conjecture that such a strong equivalence between $\oplus P$ and NT does not hold.

[5] Here a *computation* is not the sequence of instantaneous descriptions that encode the complete computation path but rather, a computation is the sequence of 0's and 1's that tell us which of the nondeterministic branch points are taken as a nondeterministic Turing machine traverses a particular computation.

To see that T is truth-table reducible to S, note first that, by assumption (iv), no string of the form $\langle x, 0^{p(|x|)}\rangle$ or $\langle x, 1^{p(|x|)}\rangle$ is in B. Using this and the fact that T is the *parity* set for M's computations, we see that for all x

$$x \in T \Leftrightarrow \text{parity}_B\,(\langle x, 0^{p(|x|)}\rangle) \neq \text{parity}_B\,(\langle x, 1^{(p|x|)}\rangle).$$

But B is simply the boundary of S, so by observation $(*)$, which preceded the statement of this theorem, we have that for $x \neq 0$,

$$x \in T \Leftrightarrow S(\langle x, 0^{p(|x|)}\rangle) \neq S(\langle x, 1^{p(|x|)}\rangle),$$

which is a very simple polynomial-time two-truth-table reduction of T to S.

But we want the stronger result that $T \leqq_1^P S$. To obtain this, we modify S by changing the definition of B. Note that if we simply *doubled* the number of accepting paths then on every computation tree we would *always* have that $\text{parity}_B(\langle x, 0^{p(|x|)}\rangle) = \text{parity}_B\,(\langle x, 1^{p(|x|)}\rangle) = 0$. This would make the membership test we have given for T always come out false, but note that if we could then always find a relevant pair $\langle x, y\rangle$ such that *exactly half* of x's computations always preceded $\langle x, y\rangle$, then we would have that $x \in T$ if and only if $\text{parity}_B\,(\langle x, y\rangle) = 1$.

This requirement is easily met. We modify S by changing the definition of the boundary B to

$$(**) \qquad B = \{\langle xc, y\rangle \colon c \in \{0, 1\} \text{ and } y \text{ is an accepting computation of } M(x)\}.$$

For each accepting computation of $M(x)$, there are now two elements of B, one less than $\langle x1, 0^m\rangle$, one greater. Furthermore, $\text{parity}_B\,(\langle x0, 0^{p(|x|)}\rangle) = \text{parity}_B\,(\langle x1, 1^{p(|x|)}\rangle) = 0$. Therefore,

$$x \in T \Leftrightarrow \text{parity}_B\,(\langle x1, 0^{p(|x|)}\rangle) = 1 \Leftrightarrow \langle x1, 0^{p(|x|)}\rangle \in S.$$

This gives the desired polynomial one-one reduction of T to S. The set S is again near-testable because B is its boundary and B in in P.

This reduction is polynomially invertible and has a polynomially decidable range because of our assumptions about p. □

COROLLARY 2.4. *$\oplus P \neq P$ if and only if $NT \neq P$.*

Note that for any one-one, polynomially computable, polynomially honest function f, the set Range (f) is in $\oplus P$. In fact Range (f) is in the subclass of $\oplus P$ called UP, which is the class of sets accepted by nondeterministic machines that always have at most one accepting computation [Va76]. It has been shown that UP is equal to P if and only if every one-one polynomially computable and polynomially honest function has a polynomially computable inverse [GS84]. Polynomially computable functions of this form that do *not* have polynomially computable inverses are called *one-way* functions. Thus we have the following corollary.

COROLLARY 2.5. *If one-way functions exist, then $NT \neq P$.*

3. Complete problems for near-testable sets. The key to completeness for Valiant's class $\#P$ is the concept of a solution-preserving reduction. A reduction is said to be *parsimonious* if it preserves not only membership in a set, but also the *number of witnesses* that a given element is in the set. If we define $\#SAT$ to be the function obtained by regarding Boolean formulas as inputs and obtaining outputs by counting the number of satisfying assignments to the Boolean formulas, then it is easily seen that $\#SAT$ is complete for $\#P$, provided the reductions used in the proof of Cook's theorem are parsimonious reductions of sets in NP to SAT. If the proof is carefully done, then these reductions are in fact easily seen to be parsimonious, one-one and

polynomially invertible ([S75]; see [MY78, Thm. 7.3.9] for a simple proof). Valiant [Va79] has shown that a number of interesting sets, including $\#SAT$, are complete for $\#P$.

There is a corresponding definition of $\oplus SAT$, which is the set of Boolean formulas that have an odd number of satisfying assignments. Furthermore, because the reductions used in the proof of Cook's theorem are parsimonious, the proof that shows that $\#SAT$ is $\leqq_1^P$-complete for $\#P$ also shows that $\oplus SAT$ is $\leqq_1^P$-complete for $\oplus P$.

Since $NT \subseteq \oplus P$, the near-testable set to which $\oplus SAT$ can be reduced using Theorem 2.3 must be $\leqq_1^P$-complete not only for the near-testable sets, but also for $\oplus P$. We call this set $NTSAT$.[6] Remember that the construction used in Theorem 2.3 implicitly defined a near-testable set by defining its boundary. If p is a polynomial bound on the run-time of the nondeterministic Turing machine recognizing $\oplus SAT$ satisfying the criteria of that construction, and $|y| = p(|x|)$, then the following hold: (1) $\langle x0, y\rangle \in NTSAT$ if and only if there are an odd number of satisfying assignments for formula x up to and including y, and (2) $\langle x1, y\rangle \in NTSAT$ if and only if the number of satisfying assignments for formula x, plus the number of satisfying assignments for x up to y, is odd.

Valiant and Vazirani [VV86] have recently shown that $\oplus SAT$ is of interest in the study of randomized reductions since it is NP-hard under randomized reductions.

DEFINITION 3.1. S is reducible to T by a randomized polynomial-time reduction r if there is a polynomial-time probabilistic Turing Machine computing r, and polynomial $p(n)$, such that for all x, if $x \notin S$ then $r(x) \notin T$, and if $x \in S$ then

$$\text{Probability}\,[r(x) \in T] \geqq p(|x|)^{-1}.$$

THEOREM 3.2. *The set NTSAT is*

(i) *Polynomially isomorphic to* $\oplus SAT$,
(ii) $\leqq_1^P$*-complete both for NT and for* $\oplus P$,
(iii) *NP-hard under randomized reductions,*[7] *and*
(iv) *Truth-table self-reducible via a simple truth-table of size two.*

Proof. (i) By definition, $\oplus SAT \leqq_1^P NTSAT$ under the one-one, polynomially *invertible* function f of Theorem 2.3. The set $\oplus SAT$ is a *polynomial cylinder* because it is easily seen to have a one-one polynomially computable, polynomially invertible, *padding function*, pad (x, y) such that for all x and y

$$x \in \oplus SAT \Leftrightarrow \text{pad}\,(x, y) \in \oplus SAT.$$

(Simply "pad" Boolean expressions by adding useless variables. If done properly, this will not change the parity of the satisfying assignments for SAT.) Since $\oplus SAT$ is $\leqq_1^P$-complete for $\oplus P$, $NTSAT \leqq_1^P \oplus SAT$. These interreducibilities between $NTSAT$ and $\oplus SAT$ imply that the padding function for $\oplus SAT$ induces a corresponding padding function for $NTSAT$, so both are cylinders. It follows (see, e.g., Lemma 2.4 of [MY85]) that $\oplus SAT$ and $NTSAT$ are polynomially isomorphic.

(ii) This follows immediately from (i) since $NT \subseteq \oplus P$, and hence $NTSAT \in \oplus P$.

(iii) First observe from the definition of randomized polynomial-time reductions that if S is reducible to B via a randomized polynomial-time reduction and if $B \leqq_1^P C$,

[6] Technically, $NTSAT$ depends both on the choice of the parity machine M, which recognizes $\oplus SAT$, and on the polynomial p, used as an explicit bound on the run time of M. For now, any choice of M and p satisfying conditions (i)–(iv) in the proof of Theorem 2.3 will do. When we use $NTSAT$ in Theorem 3.2, we will require that the machine M be chosen in a very straighforward and natural way.

[7] Richard Biegel [Be87] has independently constructed a set that is $\leqq_{2-tt}^P$-complete for NT and NP-hard under randomized reductions.

then S is reducible to C via a randomized polynomial-time reduction. But $\oplus SAT$ is hard for NP under randomized polynomial-time reductions, and $\oplus SAT \leq_1^P NTSAT$.

(iv) As we will see, this follows from a careful look at the proof of Theorem 2.3.

To this end, consider the set $NTSAT$. Recall that $\oplus SAT$ came from a standard encoding of SAT, and that the machine M that recognized SAT was required not to have any accepting choice sequences in 0^* or in 1^*. Without loss of generality, we can assume first that the formulas in $\oplus SAT$ are all in conjunctive normal form, and we can further assume by appending appropriate useless conjuncts, (e.g., "$[\neg x_i]$ & $[x_k]$" for new variables x_i and x_k), that the formulas used in defining $\oplus SAT$ have no satisfying assignments in 0^* or in 1^*.

Consider now a standard nondeterministic machine M that accepts SAT, and consider formulas with variables $x_1, x_2, \cdots, x_n$. The machine M runs in time quadratic in the length of the *formula*, and in the proof we assumed that the computation, i.e., the list of choices of branch points, always had length equal to the computational time of M. But in fact, if the machine M for SAT is chosen in its most obvious way, then the *relevant* branch points for the nondeterministic machine always correspond exactly to the choice of assignments for the variables $x_1, x_2, \cdots, x_n$. Thus for this simple case, in the proof of Theorem 2.3, we can assume that the computations, i.e., the sequence of choices at the branch points, always have length exactly n.

Returning to the proof of part (iv) of the theorem, first consider the boundary set B defined in equation (**) in the proof of Theorem 2.3. For any x there are an *even* number of relevant pairs $\langle xc, y\rangle$ that *are* in the boundary set B. Thus, since we began by putting $0 \in \overline{NTSAT}$, and since B is the boundary for $NTSAT$, we must have that for all x, $\langle x0, 0^{p(|x|)}\rangle \notin NTSAT$. Thus for the *first* computation sequence for x (i.e., the sequence whose nondeterministic guesses are all zeros), membership in $NTSAT$ is directly testable.

Now let $F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n)$ be any Boolean formula in conjunctive normal form as described above. For any such formula, we made the *possible* computation choices of the standard nondeterministic Turing machine that recognizes SAT in polynomial time and the *parity* machine that recognizes $\oplus SAT$ correspond *exactly* to the 2^n zero-one valued Boolean vectors of the form $b_1, \cdots, b_{j-1}, b_j, b_{j+1}, \cdots, b_n$, which represent *possible* satisfying assignments for $F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n)$. Thus in the proof of Theorem 2.3 equation (*), we can write the inputs and computation choices that correspond to *relevant* pairs for the set B in the form[8]

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), c, b_1, \cdots, b_{j-1}, b_j, b_{j+1}, \cdots, b_n,$$

where the b_i's and the bit c are all just Boolean values. Now for this Boolean formula F let

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), c, b_1, \cdots, b_{j-1}, b_j, b_{j+1}, \cdots, b_n$$

correspond to *any* relevant pair.

Case 1. If $c = 0$ and all $b_j = 0$, then we already know that

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), c, b_1, \cdots, b_{j-1}, b_j, b_{j+1}, \cdots, b_n \notin NTSAT.$$

Case 2. Assume that $b_1, \cdots, b_{j-1}, b_j, b_{j+1}, \cdots, b_n \neq 0^n$. Let j be the leftmost position where the bit $b_j = 1$ (j may equal 1). In this case,

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), c, 0_1, \cdots, 0_{j-1}, 1_j, b_{j+1}, \cdots, b_n$$
$$\in NTSAT \text{ if and only if}$$

[8] For ease of notation, we will drop explicit reference to the encoding $\langle xc, y\rangle$ for the rest of this proof.

there are an odd number of Boolean assignments less than or equal to $c, 0_1, \cdots, 0_{j-1}, 1_j, b_{j+1}, \cdots, b_n$ that represent successful computations on

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n).$$

But the assignments less than or equal to $c, 0_1, \cdots, 0_{j-1}, 1_j, b_{j+1}, \cdots, b_n$ are exactly those less than or equal to

$$c, 0_1, \cdots, 0_{j-1}, b_j, b_{j+1}, \cdots, b_n$$

in which the bit b_j is fixed at 1 together with those less than or equal to

$$c, 0_1, \cdots, 0_{j-1}, b_j, 1_{j+1}, \cdots, 1_n$$

in which the bit b_j is fixed at 0.

Therefore, we are able to see that there are an odd number of Boolean assignments less than or equal to $c, 0_1, \cdots, 0_{j-1}, 1_j, b_{j+1}, \cdots, b_n$ that are successful computations on $F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n)$ *if and only if* the parity of the assignments less than or equal to $c, 0_1, \cdots, 0_{j-1}, b_{j+1}, \cdots, b_n$ that are successful computations on $F(x_1, \cdots, x_{j-1}, 1, x_{j+1}, \cdots, x_n)$ is *un*equal to the parity of the assignments less than or equal to $c, 0_1, \cdots, 0_{j-1}, 1_{j+1}, \cdots, 1_n$ that are successful computations on $F(x_1, \cdots, x_{j-1}, 0, x_{j+1}, \cdots, x_n)$. That is,

$$(***)\ [\![F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), c, 0_1, \cdots, 0_{j-1}, 1_j, b_{j+1}, \cdots, b_n \in NTSAT]\!]$$
$$\Leftrightarrow [\![F(x_1, \cdots, x_{j-1}, 0, x_{j+1}, \cdots, x_n), c, 0_1, \cdots, 0_{j-1}, 1_{j+1}, \cdots, 1_n \notin NTSAT$$
$$\Leftrightarrow F(x_1, \cdots, x_{j-1}, 1, x_{j+1}, \cdots, x_n), c, 0_1, \cdots, 0_{j-1}, b_{j+1}, \cdots, b_n \in NTSAT]\!],$$

which is a standard example of a two-truth-table self-reduction of $NTSAT$ to shorter elements of $NTSAT$.

Case 3. Finally, suppose $c=1$ and all $b_j=0$. In this case, we know by our assumptions on the computations that $F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n)$, $1, 0_1, \cdots, 0_{j-1}, 0_j, 0_{j+1}, \cdots, 0_n \notin B$. But this implies that

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), 1, 0_1, \cdots, 0_{j-1}, 0_j, 0_{j+1}, \cdots, 0_n \in NTSAT \Leftrightarrow$$
$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), 0, 1_1, \cdots, 1_{j-1}, 1_j, 1_{j+1}, \cdots, 1_n \in NTSAT.$$

This is not a self-reduction since both formulas have the same length, but for the latter formula $b_j \neq 0$, so the earlier results may be used to self-reduce

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), 1, 0_1, \cdots, 0_{j-1}, 0_j, 0_{j+1}, \cdots, 0_n$$

by instead using (***) to self-reduce

$$F(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_n), 0, 1_1, \cdots, 1_{j-1}, 1_j, 1_{j+1}, \cdots, 1_n. \qquad \square$$

4. Polynomial isomorphisms and generalizations of *NT*. In this section we address the question, "To what extent is the property of being near-testable preserved under polynomial-time isomorphisms?" Given the close connection between near-testable sets and their boundaries, we might superficially expect that an isomorphism between the boundaries of two near-testable sets would induce an isomorphism of the corresponding sets, or conversely that an isomorphism between two near-testable sets would induce an isomorphism of the corresponding boundaries. But such expectations fail.

OBSERVATION 4.1. For near-testable sets S and T

(i) Boundary $(S) \cong^P$ boundary (T) does not imply that $S \cong^P T$, and

(ii) $S \cong^P T$ does not imply that boundary $(S) \cong^P$ boundary (T).

Proof. (i) If we take a very sparse polynomially decidable set, say, one whose elements satisfy $x<y$ implies $|x|<2^{|y|}$, then this single set serves as a boundary for

two near-testable sets S and $\bar{S}$. Clearly, S and $\bar{S}$ cannot be polynomially isomorphic even though they have the *same* boundary. Thus we cannot expect polynomial isomorphisms of boundaries to induce polynomial isomorphisms of near-testable sets, and no generalization of the notion of near-testability can change this.

(ii) If we ask the converse question, whether a polynomial isomorphism of near-testable sets should imply an isomorphism of their boundaries, the answer is again "no," but in this case the answer is less definitive. For example, let $S = \{0x: x \in \{0, 1\}^*\}$ and $T = \{x0: x \in \{0, 1\}^*\}$. In this case S and T are trivially near-testable since they are in P. Note that $S \cong^P T$. But the boundary of T is all of $\{0, 1\}^*$, and the boundary of S contains only two elements of each length, so it is polynomially sparse. Therefore boundary $(S) \not\cong^P$ boundary (T). □

Note that in our second example, the boundaries failed to be polynomially isomorphic because of the rigidity of the underlying ordering structure that we insisted on using for our notion of "nearness." If we had been allowed to use *reverse* lexicographic ordering to measure "nearness" in the set T, then the boundaries of S and T in the second example would have been polynomially isomorphic.

One property that distinguishes near-testable and wdq self-reducible sets from other more general self-reducible sets is that their definition is based on the standard lexicographic ordering of Σ^*. For this reason near-testability and other notions of self-reducibility are quite sensitive to the encodings of the set with respect to this fixed-order structure. For example, these properties of sets are not necessarily closed under polynomially computable isomorphisms (or even length-preserving permutations) of Σ^*. Thus, because of our reliance on the canonical lexicographic ordering, the property of near-testability as we have defined it is probably not preserved under polynomial isomorphisms.

OBSERVATION 4.2. If $NT \neq P$, then near-testability is not preserved by polynomial isomorphisms.

Proof. Since $NTSAT$ is complete for NT, if $NT \neq P$ then $NTSAT \notin P$. As observed in the proof of Theorem 3.2, $NTSAT$ has a polynomially computable padding function pad. Let $T = \{x0: x \in NTSAT\}$. It is easy to see that T and $NTSAT$ are polynomially many-one equivalent and that the padding function pad for $NTSAT$ can be used to induce a corresponding padding function on T. Thus both $NTSAT$ and T are *polynomial cylinders*, and, as in the proof of Theorem 3.2, it follows from Lemma 2.4 of [MY85] that $NTSAT$ and T are polynomially isomorphic. Thus $T \notin P$. We now show that T is not near-testable. Since every other string is *not in* T, we see that $x \in T$ if and only if $x \in$ boundary $(T) \cap \{\Sigma^*0\}$. Therefore boundary $(T) \in P \Rightarrow T \in P$. Thus boundary $(T) \notin P$, so T is *not* near-testable. □

Our last two observations suggest that, if we want the property of being near-testable to be more broadly applicable and to be closed under polynomial isomorphisms, then we should relax our requirement that "nearness" be measured only in terms of the standard lexicographic ordering of Σ^*. Note that similar problems arise for properties like Turing self-reducibility when the definition of the property is strictly tied to the lexicographic ordering of Σ^*. The situation can be remedied by allowing more general orderings of Σ^*; for example, Meyer and Paterson [MP79] and Ko [Ko83] have given generalized definitions for underlying ordering structures for use in defining Turing self-reducibility. We will use a similar approach to define a broader class, NT^*, that has many of the properties of NT, yet is more robust.

We would like to say that a set is in NT^* if there is *some suitable polynomial ordering* relative to which the set is near-testable. To formalize this definition we must describe the properties of a suitable ordering.

Our orderings will be *tree orderings*, or more precisely *forests*, where the minimal elements are the roots of the trees. An element x is *less than* an element y, $x < y$, if $x \neq y$ and x and y lie on the same branch of a tree with x closer to the root than y. The roots of the trees must form a polynomially decidable set, which we will call *ROOTS*. We will say that such orderings are *polynomially computable* if $<$ is a polynomially testable relation and if, for $x \notin ROOTS$, the function pred (x) = *the unique element* y *such that* $y < x$ *and for all* z, $z < x \Rightarrow z \leq y$, is polynomially computable. (It follows that $ROOTS \in P$, so, without loss of generality, we will assume that pred (x) crisply detects when its input is a root.)

In addition, to preserve some of the time and space complexity properties of NT we will require that the ordering be *exponentially well-founded.* For some polynomial p, $\max\{|z|: z < x\} \leqq p(|x|)$. This will imply that $|\{y: y < x\}| \leqq 2^{p(|x|)}$, and thus that for any x, the path from x back to its root behaves reasonably like the sequence x, $x-1, \cdots, 1$.

DEFINITION 4.3. A set S is in NT^* if there exists a polynomially computable, exponentially well-founded ordering, $<_S$, such that

(i) $ROOTS \cap S$ is decidable in polynomial time, and

(ii) For $x \notin ROOTS$, $t(x) = \chi_S(x) + \chi_S(\text{pred}_S(x)) \pmod 2$ is a polynomially computable function. For a set S in NT^* we define boundary $(S) = \{x$: exactly one of x and $\text{pred}_S(x) \in S\}$.

THEOREM 4.4. (i) $NT \subseteq NT^* = coNT^* \subseteq \oplus P$.

(ii) $NT^* \neq P \Leftrightarrow NT \neq P \Leftrightarrow NT^* \neq NT$.

Proof. (i) Any set in NT is in NT^* since the lexicographic ordering satisfies the properties of Definition 4.3. NT^* is closed under complements since the tree defining a set $S \in NT^*$ can be switched to define $\bar{S}$ simply by noting that if $ROOTS \cap S$ is decidable in polynomial time, so is $ROOTS \cap \bar{S}$. Sets in NT^* are in $\oplus P$ for essentially the same reason that sets in NT are in $\oplus P$: we can design a nondeterministic machine that on input x, *guesses* elements z along the branch of a tree from a root to x, checks that $z \leqq_S x$, calculates $y = \text{pred}(z)$ if z is not a root, and checks (using the polynomially computable function t) that y is a boundary element along this path or, if z is a root, checks that $z \in S$. Thus the computation tree for this machine on input x has accepting paths for each element z along the branch from x to its root for which $z \in S \Leftrightarrow \text{pred}(z) \notin S$, plus one additional accepting path if the root is in S. Thus $x \in S$ if and only if the number of accepting paths is *odd.*

(ii) If $NT^* \neq P$, then $\oplus P \neq P$, so, by Corollary 2.4, $NT \neq P$. If $NT \neq P$, the image, T, of $NTSAT$ in Observation 4.2 is an example of a set in $NT^* - NT$. If $NT^* \neq NT$ then trivially $NT^* \neq P$. □

From the above result we have that $P \subseteq NT^* \subseteq PSPACE$, as well as the following corollary.

COROLLARY 4.5. *Every set in* $\oplus P$ *is* $\leqq_1^P$*-reducible to a set in* NT^*.

One motive for extending the definition of NT was to find a superset of NT that was closed under polynomial isomorphisms. The next theorem shows that NT^* accomplishes this goal.

THEOREM 4.6. NT^* *is closed under polynomial isomorphisms.*

Proof. Suppose that $S \in NT^*$ and g is a polynomial isomorphism between S and a set T, $g: T \to S$. We will denote by $<_S$, pred_S, $ROOTS_S$, and t_S the tree ordering, predecessor function, set of *roots* and near-testability function for S. We define similar relations, sets, and functions for T as follows.

(i) $x <_T y$ *if and only if* $g(x) <_S g(y)$,

(ii) $\text{pred}_T(x) = g^{-1} \circ \text{pred}_S \circ g(x)$,

(iii) $ROOTS_T = \{x: g(x) \in ROOTS_S\}$, and
(iv) $t_T(x) = t_S(g(x))$.

Clearly, the ordering $<_T$ is polynomially computable, $ROOTS_T$ is a polynomially decidable set of roots for $<_T$, and t_T is a near-testability relation for T. Also clearly, $ROOTS_T \cap T$ is polynomially decidable. To see that $<_T$ is exponentially well-founded, let p be the polynomial that witnesses that $<_S$ is exponentially well-founded and let p_g be a polynomial that bounds the stretching and shrinking done by g. Then

$$\max\{|z|: z <_T x\} \leqq p_g(\max\{|z|: g(z) <_S g(x)\}) \leqq p_g(p(p_g(|x|))).$$

In addition, note that

$$|\{y: y <_T x\}| = |\{y: y <_S g(x)\}| \leqq 2^{p(|g(x)|)} \leqq 2^{p(p_g(|x|))}. \qquad \square$$

In the calculations above, we never used the fact that the inverse $g^{-1}(x)$ is unique. All we needed was that, for each x, the set $g^{-1}(x)$ can be found in polynomial time and that its size is polynomially related to the length of x. Therefore, if g is not one-one but is *polynomially many-one* and completely polynomially invertible in the sense just described,[9] then for each x we can $<_T$ order the elements of $g^{-1}(x)$ by simply using the natural lexicographic ordering. This yields the following observation.

OBSERVATION 4.7. NT^* is closed under polynomial many-one reductions that are onto Σ^* and have completely polynomially computable inverses.

It is also worth observing that the isomorphisms of Theorem 4.6 preserve the tree structures of the underlying orderings. Theorem 3.2(i) tells us that the standard complete set for $\oplus P$, $\oplus SAT$, is polynomially isomorphic to $NTSAT$. Thus we have Corollary 4.8.

COROLLARY 4.8. *The standard complete set for $\oplus P$, $\oplus SAT$, is in NT^* with an underlying polynomial ordering of type $(N, <)$.*

Observation 4.1(ii) showed that polynomial isomorphisms of sets in NT do not imply the existence of isomorphisms between the boundaries of those sets. This result depended on the canonical ordering $(N, <)$ in the definition of NT. For NT^*, the situation is different. In Theorem 4.6 we saw that if we have a set in NT^* based on the exponentially well-founded tree ordering $<_S$, then *any* isomorphism g provides an isomorphism of the *induced* ordering $<_T$ (as described in the proof of Theorem 4.6), and it is obvious that the boundaries under $<_S$ and $<_T$ are polynomially isomorphic. We state this result as Corollary 4.9.

COROLLARY 4.9. *Given any two polynomially isomorphic sets S and T in NT^*, the boundary of S is isomorphic to the boundary of T. (The boundaries are taken with respect to the tree ordering induced by the ismorphism.)*

Corollary 4.8 raises the possibility that $\oplus P = NT^*$. We conjecture that the containment $NT^* \subseteq \oplus P$ is proper. Except for the obvious hypothesis that there exist sparse sets in $\oplus P - P$, we do not have interesting (and nontrivial) conditions that imply that this is true, although in § 5 we will see that with respect to a random oracle the containment is proper with probability one.

If the containment is not proper, then $NT^* = \{S: S \leqq_m^P NTSAT\}$, and this would say that this generalized notion of near-testability is not just an interesting ordering property that some sets possess, but that it defines a natural complexity class.

To prove the oracle results in § 5, the following variation of Observation 1.2(ii) will be useful.

---

[9] Functions with this property were called *strongly invertible* by Allender and Rubinstein [AR88]. Such functions will be discussed in more detail in § 6.

OBSERVATION 4.10. If $S \in NT^*$ and if either S or $\bar{S}$ is polynomially sparse, then $S \in P$.

Proof. The proof is the same as for NT. Assume that S is sparse, and let p and q be polynomials such that of the elements of length less than or equal to n, at most $q(n) \in S$, and of the elements preceding x in the ordering $<_S$ none is larger than $p(|x|)$. Then for any x, consider the $2 * q(p(|x|)) + 1$ elements in the chain of elements that immediately precede x in the ordering $<_S$. If there is an element of $ROOTS$ within $2 * q(p(|x|)) + 1$ "steps" of x, then membership of x in S can be determined directly. Otherwise, these $2 * q(p(|x|)) + 1$ elements may be split into two subsets by enumerating them using the predecessor function and splitting them every time the near-testability relationship tells us that we have crossed a border between S and $\bar{S}$. One of these subsets will be contained in S and the other in $\bar{S}$. Because S is sparse, the larger of these subsets is in $\bar{S}$. This gives a polynomial test for membership in S. Obviously if $\bar{S}$ is sparse, then the same test works, except that the larger of the two groups of elements in the chain of predecessors is in S. □

5. Relativizing *NT*. Bennett and Gill [BG81] began the study of results that hold for almost every oracle. Although there are examples of results that hold for almost every oracle yet are false in an unrelativized setting [Ku82], probability one results still may be of some interest—in some sense, they explore the power of computation in the presence of freely available randomness.

In this section we combine relativized versions of our results above with the results of Bennett and Gill, and of Regan and Royer, to show that, with probability one, NT^A contains computationally difficult sets. We also show that, with probability one, both NT^A and NT^{*A} are incomparable with NP^A and with $coNP^A$.

THEOREM 5.1. *Relative to a random oracle A, $P^A \subsetneq NT^A \subsetneq NT^{*A} \subsetneq \oplus P^A$, with probability one.*

Proof. Bennett and Gill [BG81, Thm. 3] use the language

$$ODD^A = \{x\text{: an odd number of strings of length } |x| \text{ are in } A\}$$

to show that $PP^A \subsetneq PSPACE^A$ with probability one. Their proof is in error [Be88], but the remaining, correct techniques of Bennett and Gill, particularly their Lemma 1, easily suffice to show that ODD^A separates $\oplus P^A$ from P^A with probability one; alternatively, Aspnes, Beigel, Furst, and Rudich have recently announced that Theorem 3 of Bennett and Gill is a valid claim, and that they indeed even have reestablished the claim, implicit in [BG81], that ODD^A witnesses the probability one separation of $\oplus P^A$ from PP^A [Be90]. If we let $T^A =_{\text{def}} 0^* \cap ODD^A$, then $ODD^A \equiv_m^P T^A$, so we now have that T^A separates P^A from $\oplus P^A$ with probability one. But T^A is sparse, and since Observation 4.10 clearly relativizes to an oracle computation, $T^A \in NT^{*A} \Leftrightarrow T^A \in P^A$. Thus $NT^{*A} \subsetneq \oplus P^A$ with probability one.

Now the proof of Theorem 2.3 and hence Corollary 2.4 also relativizes, as does Theorem 4.4. Corollary 2.4 guarantees that any oracle that separates P from $\oplus P$ also separates P from NT. And then Theorem 4.4(ii) guarantees that any oracle that separates P from NT also separates NT from NT^*. This establishes for any oracle A for which ODD^A separates P^A and $\oplus P^A$,

$$P^A \subsetneq NT^A \subsetneq NT^{*A} \subsetneq \oplus P^A.$$

Since the collection of oracles A for which ODD^A separates P and $\oplus P$ has measure one, this establishes the theorem. □

THEOREM 5.2. *With probability one relative to a random oracle A, $NT^A - PH^A \neq \emptyset$, where PH represents the polynomial hierarchy.*

Proof. Theorem 2.3 of this paper shows that every set in $\oplus P$ is polynomial-time many-one reducible to a set in *NT*. Regan and Royer [RR90] have shown that with probability one, $\oplus P^A \supset PH^A$. The theorem follows immediately from these two facts and the fact that $\oplus P$ is closed downwards under polynomial-time many-one reductions. □

THEOREM 5.3. *Relative to a random oracle A,*

$$(NP^A - NT^{*A}) \neq \varnothing \quad \textit{and} \quad (coNP^A - NT^{*A}) \neq \varnothing$$

with probability one.

Proof. Given a set A, define a function $\xi(x)=$ the string of 0's and 1's of length $|x|$ such that the kth bit is 1 if and only if $x10^{k-1} \in A$. Bennett and Gill show that, with probability one, neither the language $RANGE3^A =_{\text{def}} \{x\colon \exists y[\xi(y)=xxx]\}$ nor the language $\overline{RANGE3^A}$ contains an infinite polynomially decidable set. Since $RANGE3^A$ is obviously in NP^A, this shows that, with probability one, $RANGE3^A$ separates NP^A from P^A [BG81, Thm. 6].

But since $RANGE3^A$ and $\overline{RANGE3^A}$ are both *P*-immune, 0^* must intersect both $RANGE3^A$ and $\overline{RANGE3^A}$ infinitely often. Thus, as pointed out by Bennett and Gill, $T^A =_{\text{def}} RANGE3^A \cap 0^*$ must be a polynomially sparse set that is in $NP^A - P^A$ with probability one. Since it is polynomially sparse, just as in the proof of Theorem 5.1, T^A must therefore be in $NP^A - NT^{*A}$ with probability one.

Since NT^{*A} is closed under complements, we may separate $coNP^A$ from NT^{*A} with probability one by using the complement of T^A. □

6. Some additional facts about near-testable sets. In this final section we present additional results concerning near-testable sets. In § 1 we noted that any polynomially sparse set that is near-testable is polynomially decidable. Here we return to this theme, first discussing the effects that the distribution and density of elements have on the complexity of a near-testable set. Next, having observed in Corollary 2.5 that the existence of one-way functions implies that there are near-testable sets that are not polynomially decidable, we prove a partial converse to this result. Finally, we briefly relate *P*-selective and near-testable sets.

We begin this section with some observations on the fragility of *NT*. Because *near-testability* is sensitive to the distribution of elements in a set, and to density, certain sets are unlikely to be near-testable. For instance, since the primes (with the exception of 2) are distributed only throughout the *odd* integers, an odd number greater than 3 is prime if and only if it is in boundary ({*primes*}).

OBSERVATION 6.1. If $\{q\colon q \text{ is prime}\}$ is near-testable, then primality testing can be done in polynomial time.

Since the set of primes is known to be in *ZPP*, this tells us that not all sets in *ZPP* are near-testable unless primality testing is in *P*. This again points out how sensitive *NT* is to the underlying order structure. We can generalize this example by replacing the set of odd numbers by any polynomially recognizable set that is sufficiently dense.

DEFINITION 6.2. Let $<$ be any polynomially computable, exponentially well-founded ordering. We say that a set D is uniformly dense with respect to $<$ if there is a polynomial $p(n)$ such that for any string x, there is an element of D or a root within $p(|x|)$ "steps" preceding x in the ordering $<$.[10]

OBSERVATION 6.3. Suppose that $S \in NT^*$ via the ordering $<$. If there is a set D, uniformly dense with respect to $<$, such that $D \in P$ and $D \cap S \in P$, then $S \in P$.

[10] An inverse of this notion, that of a set being *uniformly sparse* in the standard lexicographic ordering is used in [HIS85] and also (in the uniform version that we use here in Theorem 6.7) in [GJY87c].

Proof. Suppose the polynomial $p(n)$ bounds the number of steps from any string of length n to the nearest element of D. Suppose also that both D and $D \cap S$ can each be polynomially decided. To decide whether $x \in S$, we enumerate the $p(|x|)$ strings immediately preceding x, if that many exist, and we then run the decision procedure for D on each of these strings. Once we find $y \in D$, we quickly decide whether $y \in D \cap S$. We then use the near-testability algorithm to decide membership for all of the strings from y to x, including x. □

The next observation shows that if the boundary of an NT^* set is uniformly dense then the set and its complement are many-one interreducible.

OBSERVATION 6.4. Suppose that $S \in NT^*$ and that boundary (S) is uniformly dense with respect to $<_S$; then $S \leqq_m^P \bar{S}$. (Thus, if S is in NP or in $coNP$, then $S \in NP \cap coNP$.)

Proof. Since boundary $(S) \in P$ and is uniformly dense, both S and $\bar{S}$ are infinite and in NT^*. Define a polynomial many-one reduction, f, from S to $\bar{S}$ as follows. Given x, we can find (in polynomial time) the first $y \leqq_S x$ such that $y \in$ boundary (S) or $y \in ROOTS$. If $y \in ROOTS$, then $x \in S$ if and only if $y \in S$, and for $y \in ROOTS$ this is polynomially decidable. Given $a \in S$, $b \notin S$, let

$$f(x) = \begin{cases} a, & \text{if } y \in ROOTS \cap S, \\ b, & \text{if } y \in ROOTS \cap \bar{S}, \\ \text{pred}\,(y) & \text{otherwise.} \end{cases}$$ □

The results in § 5 show that it is not likely that all sets in NP and $coNP$ are near-testable or even in NT^*. The question of whether there are *any* near-testable sets that are not in P was addressed in § 2, where we showed that $NP \neq P$ if and only if $\oplus P \neq P$. As pointed out there, the existence of one-way functions implies that UP, and hence $\oplus P$, is not equal to P. Thus, if one-way functions exist, $NT \neq P$.

A natural question is whether we actually need the existence of one-way functions in order to prove that there are sets in NT or in $\oplus P$ that are not in P. One result which suggests that one-way functions are not needed is the existence of oracles relative to which one-way functions do not exist but $P^A \neq \oplus P^A$ (this follows by the obvious modification to [Ra82, Thm. 4]). Our next result, however, gives a partial answer pointing in the other direction. Instead of asking about polynomially invertible one-one functions, we instead ask about *strongly invertible* many-one functions. Recall that in the proof of Observation 4.7 we used inverses of polynomially many-one, polynomially computable, polynomially honest functions. If f is such a function, let $f^{-1}(x) = \{y: f(y) = x\}$. Note that if f is polynomially many-one, it is conceivable that, given x, we can find $\{y: f(y) = x\}$ in time polynomial in $|x|$. Functions for which we can always complete the listing of all of $\{y: f(y) = x\}$ in time polynomial in $|x|$ are called (many-one) *strongly invertible.* Functions for which we cannot always completely list $\{y: f(y) = x\}$ in polynomial time are said to be *one-way.*

Many-one, one-way functions have been studied in other contexts, and are discussed extensively in [AR88]. For example, for strong invertibility on 0^*, Allender and Rubinstein prove that the following are equivalent:

(i) There is an honest polynomially many-one function computable in polynomial time that is not strongly invertible on 0^*.

(ii) There is a polynomially sparse set in Few $P - P$.

(iii) $E^{\text{sparse(Few } P)} \neq E$.

(iv) There is a polynomially sparse set in P that is not P-printable.

(v) There is a polynomially sparse set in deterministic logspace that is not P-printable.

In this context, we have the following partial converse to Corollary 2.5. (Recall that any set that is polynomially sparse has a polynomially sparse boundary, but the converse is not true. Note also that any *polynomially* well-founded ordering $\leqq$ will have a polynomially computable function r which finds the roots of $\leqq$.)

THEOREM 6.5. *If there is a set $S \in NT^* - P$ with a polynomially sparse boundary, and if there is a polynomial-time computable function $r(x)$ such that $r(x) \leqq x$ and $r(x) \in ROOTS$ for all x, then there is a polynomially many-one, polynomially computable, polynomially honest function that is not strongly invertible on 0^*.*

Proof. Let $B = \text{boundary}(S)$. Let $q(n)$ be a polynomial bound on the census function for B, and let $<$ be the underlying tree ordering used in witnessing that $S \in NT^*$. We define

$$f(x) = \begin{cases} 0^{|x|} & \text{if } x \in B, \\ x & \text{otherwise.} \end{cases}$$

Since $S \in NT^*$, $B \in P$, so $f(x)$ is polynomially computable. Since $|f(x)| = |x|$, f is honest. Note that $f^{-1}(0^n)$ has at most $q(n)+1$ elements, all of length n. Suppose f^{-1} *were* polynomially computable on 0^*. Given x, in time polynomial in $|x|$, we could then compute all of $f^{-1}(0), f^{-1}(0^2), \cdots f^{-1}(0^{|x|})$, throwing away any z's found in some $f^{-1}(0^n)$ $(n \leqq |x|)$ for which $z \in 0^*$ but $z \notin B$, and throwing away any z for which $z \not< x$. Thus, in time polynomial in $|x|$, we can find all z's $\leqq x$ that lie on the boundary of S. We then compute $r(x)$, and decide (in polynomial time, since $r(x) \in ROOTS$), if $r(x) \in S$. Given this information, and the number of boundary elements $z \leqq x$, we can decide $x \in S$ in polynomial time. Thus, if $S \notin P$, f must be a one-way function.

(Note that if S has at most one element of any given length, then f is a *one-one* one-way function.) □

The proof of the previous theorem requires a set in $NT^* - P$ with a polynomial-time computable function r, and a sparse boundary. Given a one-way function, Corollary 2.4 and Theorem 2.3 let us construct sets in $NT - P$. However, these sets do *not* have sparse boundaries, and we do not know interesting conditions that imply the existence of sets in $NT - P$ with sparse boundaries.[11] We can construct, although we will not do so here, an oracle A relative to which there exist sets in $NT^A - P^A$ that have boundaries with at most one element of any given length.

In closing, we give one more method that may construct sets that are *near-testable* but are not polynomially decidable. This construction not only gives additional evidence that $NT \neq P$, but it shows that the combination of near-testability and P-selectivity is unlikely to guarantee that sets are decidable in polynomial time. This is in contrast to the case for p-cheatability, since it can be shown that near-testable sets that are (2^k *for* k) p-cheatable are all decidable in polynomial time [GJY87c].[12]

To do our final construction, we need to assume the existence of *very* sparse sets in P that are not *P-printable.*

DEFINITION 6.6. A set S is uniformly log*-sparse if for all x and y in S, $x < y$ implies $2^{|x|} < |y|$.

[11] Recently Cai and Hemachandra have shown that every set in Few$P$ is in $\oplus P$ [CH]. We might hope to use this result together with the reduction from sets in $\oplus P$ to sets in NT given in Theorem 2.3 to strengthen Theorem 6.5. Unfortunately this does not seem possible because Cai and Hemachandra's result increases the number of accepting paths so that it is no longer polynomially bounded.

[12] For information about P-selective sets, see [S79], [S82a], [S82b], [Ko83], [GJY87a], and [GJY87c]. Amir, Beigel, and Gasarch [ABG87] have independently shown this last result for the special case of (2 for 1) p-cheatability.

THEOREM 6.7. *If there is a uniformly* $\log^*$*-sparse set in P that is not P-printable, then there is a P-selective near-testable set that is not polynomially decidable.*

Proof. Let S be a uniformly $\log^*$-sparse set in P that is not *P-printable*. Define a rapidly growing function f by $f(0)=2$ and $f(n+1)=2^{f(n)}$. We will let I_n be the interval of strings of length $f(n)$ up to length $f(n+1)$. From the definition of uniformly $\log^*$-sparse sets, we know that $|S \cap I_n| \leqq 1$ for each n.

Let

$$S_0 = S \cap \cup_n \{I_{3n}\}, \quad S_1 = S \cap \cup_n \{I_{3n+1}\}, \quad S_2 = S \cap \cup_n \{I_{3n+2}\}.$$

Then $S_i \in P$ for each i, and for at least one i, S_i is not P-printable. Without loss of generality, we will assume that S_0 is not P-printable.

We then define the desired set T as follows:

$$T = \{x : x \in I_{3n} \cup I_{3n+1} \text{ and } \exists y \in I_{3n} \cap S_0, y \leqq x\} \cup \{I_{3n+2} - \{f(3n+3)-1\}\}.$$

In other words, all but the last element of I_{3n+2} is always in T, and the last element of I_{3n+2} is always a boundary point. If there is an element $y \in I_{3n} \cap S_0$, then this y serves as a "breakpoint," dividing $I_{3n} \cup I_{3n+1}$ into a left-hand subinterval in $\bar{T}$ and a right-hand subinterval in T. If there is no such element, then both I_{3n} and $I_{3n+1} \subset \bar{T}$.

The boundary of T thus consists of the rightmost elements of the I_{3n+2}'s, the element of $I_{3n} \cap S_0$ when there is one, plus the left-hand endpoints of the I_{3n+2}'s for which $I_{3n} \cap S_0 = \emptyset$. By the definition of the I_k's, if y is the left-hand endpoint of I_{3n+2}, then in time polynomial in $|y|$, we can test whether there is an element of S_0 in I_{3n}. Thus, boundary $(T) \in P$, so T is near-testable.

Assuming $x \leqq y$, the following function is a *P-selection* function for T:

$$s(x,y) = \begin{cases} y & \text{if } x = f(3m+3)-1 \text{ for some } m, \text{ else,} \\ x & \text{if } x \in I_{3m+2} \text{ for some } m, \text{ else,} \\ y & \text{if } x, y \in I_{3m} \cup I_{3m+1} \cup I_{3m+2} \text{ for some } m, \text{ else,} \\ x & \text{if } x \in I_{3m} \cup I_{3m+1}, y \in I_{3n} \cup I_{3n+1} \cup I_{3n+2} \\ & \quad \text{for some } n > m \text{ and } x \in T, \text{ else,} \\ y & \text{if } x \in I_{3m} \cup I_{3m+1}, y \in I_{3n} \cup I_{3n+1} \cup I_{3n+2} \text{ for some } n > m \text{ and } x \notin T. \end{cases}$$

Note that in the last two cases we can test $x \in T$ in time polynomial in $|y|$.

Finally, suppose that $T \in P$. Then the following algorithm *P-prints* S_0: for each k, if $0^k \in T$ and $0^{k+1} \notin T$, then there is exactly one boundary point for T of length k. The assumption that $T \in P$ thus enables us to find this boundary point in polynomial time by using binary search. By definition of T, any boundary point for T has one of three forms, and two of these, the elements of the form $f(3n+3)-1$ and of the form $f(3n+2)$ simply are not in S_0. Any other boundary point is in S_0. Since *all* elements of S_0 are boundary points of T, this gives a polynomial algorithm for printing S_0, a contradiction.

Thus, T is a *P-selective*, *near-testable* set that is not in P. □

7. Conclusions. This paper introduces the *near-testable* sets, a class of sets having a particularly simple internal ordering structure. These sets are defined to capture one notion of having an efficient partial membership test; in polynomial time, we can determine whether exactly one of a string and its predecessor are in a near-testable set. Though the class of sets that have efficient exact membership tests, P, seems quite small, we have shown that an apparently far broader class of sets are near-testable: all $\oplus P$ sets $\leqq_1^P$ reduce to near-testable sets. Thus, our results precisely locate the complexity of near-testable sets in terms of an important and well-studied class, $\oplus P$. It follows from our results that there are near-testable sets that are not polynomially

decidable if and only if there are sets in $\oplus P$ that are not polynomially decidable, and that with probability one relative to a random oracle, near-testable sets separate *PSPACE* from the polynomial hierarchy.

Acknowledgments. The authors would like to thank Eric Allender, Klaus Ambos-Spies, José Balcázar, Sanjay Jain, and Alan Selman for helpful discussions and suggestions, and Richard Beigel and Kenneth Regan for allowing us to discuss their work in progress.

REFERENCES

[ABG87] A. AMIR, R. BEIGEL, AND W. GASARCH, personal communication, 1987.

[AR88] E. ALLENDER AND R. RUBINSTEIN, *P-printable sets*, SIAM J. Comput., 17 (1988), pp. 1193-1202.

[BGS75] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the P = ? NP question*, SIAM J. Comput., 4 (1975), pp. 431-442.

[Ba87] J. BALCÁZAR, *Self-reducibility*, in Proc. Symposium on the Theory of Automata and Computing, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag, Berlin, New York, 1987, pp. 136-147; J. Comput. System Sci., 41 (1990), pp. 367-388.

[Be88] ———, *Relativized counting classes: Relations among thresholds, parity, and mods*, Tech. Report 88-09, Department of Computer Science, The Johns Hopkins University, Baltimore, MD, 1988. J. Comput. System Sci., to appear.

[Be87] R. BEIGEL, personal communication, 1987.

[Be90] ———, personal communication, 1990.

[BG81] C. BENNETT AND J. GILL, *Relative to a random oracle A, $P^A \neq NP^A$ with probability one*, SIAM J. Comput., 10 (1981), pp. 96-113.

[CH90] J. CAI AND L. HEMACHANDRA, *On the power of parity polynomial time*, Math. Systems Theory, 23 (1990), pp. 95-106.

[GJ79] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, New York, 1979.

[Go89] J. GOLDSMITH, *Polynomial isomorphisms and near-testable sets*, Ph.D. thesis, Tech. Report 816, University of Wisconsin, Madison, WI, 1989.

[GHK90] J. GOLDSMITH, L. HEMACHANDRA, AND K. KUNEN, *On the structure and complexity of infinite sets with minimal perfect hash functions*, Tech. Report 339, University of Rochester, Rochester, NY, 1990, pp. 1-16.

[GJY87a] J. GOLDSMITH, D. JOSEPH, AND P. YOUNG, *Self-reducible, p-selective, near-testable, and p-cheatable sets: The effect of internal structure on the complexity of a set, preliminary abstract*, in Proc. 2nd Annual Structure in Complexity Theory Conference, 1987, pp. 50-59. Also in more complete form as Tech. Report 87-06-02, University of Washington, Seattle, WA, 1987, and as Tech. Report 743, University of Wisconsin, Madison, WI, 1987, pp. 1-22.

[GJY87b] ———, *A note on bi-immunity and p-closeness of p-cheatable sets in P/poly*, Tech. Report 87-11-05, University of Washington, Seattle, WA, 1987, and Tech. Report 741, University of Wisconsin, Madison, WI, 1987, pp. 1-13; J. Comput. System Sci., to appear.

[GJY87c] ———, *Using self-reducibilities to characterize polynomial time*, Tech. Report 87-11-11, University of Washington, Seattle, WA, 1987, and Tech. Report 749, University of Wisconsin, Madison, WI, 1987, pp. 1-20; Inform. and Comput., to appear.

[GS84] J. GROLLMAN AND A. SELMAN, *Complexity measures for public key cryptosystems*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1984, pp. 495-503; SIAM J. Comput., 17 (1988), pp. 309-335.

[GP86] L. GOLDSCHLAGER AND I. PARBERRY, *On the construction of parallel computers from various bases of Boolean functions*, Theoret. Comput. Sci., 43 (1986), pp. 43-58.

[HIS85] J. HARTMANIS, N. IMMERMAN, AND V. SEWELSON, *Sparse sets in NP − P: EXPTIME versus NEXPTIME*, Inform. and Control, 65 (1985), pp. 159-181.

[H90] L. HEMACHANDRA, *Algorithms from complexity theory: Polynomial-time operations for complex sets*, in Proc. SIGAL International Symposium on Algorithms, Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 450 (1990), pp. 221-231.

[HH90] L. HEMACHANDRA AND A. HOENE, *On sets with efficient implicit membership tests*, in Proc. Fifth Annual Structure in Complexity Theory Conference, IEEE Computer Society, Washington DC, 1990, pp. 11-19; SIAM J. Comput., to appear.

[HHSY] L. HEMACHANDRA, A. HOENE, D. SIEFKES, AND P. YOUNG, *On sets polynomially enumerable by iteration*, Theoret. Comput. Sci., to appear.

[Ko83] K. KO, *On self-reducibility and weak P-selectivity*, J. Comput. System Sci., 26 (1983), pp. 209-221.

[Ku82] S. KURTZ, *On the random oracle hypotheses*, in Proc. 14th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 224-230; Inform. and Control, 57 (1983).

[MY78] M. MACHTEY AND P. YOUNG, *An Introduction to the General Theory of Algorithms*, Elsevier, New York, 1978, pp. 1-264.

[MY85] S. MAHANEY AND P. YOUNG, *Reductions among polynomial isomorphism types*, Theoret. Comput. Sci., 39 (1985), pp. 207-224.

[MP79] A. MEYER AND M. PATERSON, *With what frequency are apparently intractable problems difficult?* MIT/LCS/TM-126, Massachusetts Institute of Technology, Cambridge, MA, 1979.

[PZ83] C. PAPADIMITRIOU AND S. ZACHOS, *Two remarks on the power of counting*, in Proc. 6th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 145, Springer-Verlag, Berlin, New York, 1983, pp. 269-275.

[Ra82] C. RACKOFF, *Relativized questions involving probabilistic algorithms*, J. Assoc. Comput. Mach., (1982), pp. 261-268.

[RR90] K. REGAN AND J. ROYER, personal communication, 1990.

[S79] A. SELMAN, *P-selective sets, tally languages, and the behavior of polynomial reducibilities on NP*, Math. Systems Theory, 13 (1979), pp. 55-65.

[S82a] ———, *Analogues of semi-recursive sets and effective reducibilities to the study of NP complexity*, Inform. and Control, 52 (1982), pp. 36-51.

[S82b] ———, *Reductions on NP and P-selective sets*, Theoret. Comput. Sci., 19 (1982), pp. 287-304.

[S75] J. SIMON, *Some central problems in computational complexity*, Ph.D. thesis, Tech. Report TR75-224, Cornell University, Ithaca, NY, 1975.

[Va76] L. VALIANT, *The relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20-23.

[Va79] ———, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410-421.

[VV86] L. VALIANT AND V. VAZIRANI, *NP is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85-93.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 524-536, June 1991

© 1991 Society for Industrial and Applied Mathematics
009

COMPRESSION AND RANKING*

ANDREW V. GOLDBERG† AND MICHAEL SIPSER‡

Abstract. A complexity-theoretic approach to the classical data compression problem is presented. A notion of language compressibility is defined, and it is shown that essentially all strings in a sufficiently sparse "easy" (e.g., polynomial-time) language can be compressed efficiently. A notion of ranking as a form of optimal compression is also defined, and it is shown that some "very easy" languages (e.g., unambiguous context-free languages) can be ranked efficiently. Languages that cannot be compressed or ranked efficiently under various complexity-theoretic assumptions are exhibited.

The notion of compressibility is closely related to Kolmogorov complexity and randomness. This relationship and the complexity-theoretic implications of our results are discussed.

Key words. data compression, Kolmogorov complexity, computational complexity

AMS(MOS) subject classification. 68

1. Introduction. In the classical data compression problem one is given a long string and wishes to obtain a succinct representation of it so that it can be transmitted or stored more efficiently. It is important that both the process of coding, i.e., mapping the string to its representation, and the decoding process have low computational complexity. This problem is typically solved in practice by exploiting characteristics of the language of source strings, such as the low Shannon entropy of English text, to achieve the compressed representation. The goal of this paper is to show that, under certain very general complexity-theoretic assumptions about a language, one may efficiently compute a compressed representation for essentially all of its members.

Data compression is closely related to Kolmogorov complexity [K]. The Kolmogorov complexity of a string is the length of the shortest program that generates the string. The shortest program may be viewed as a succinct representation. Kolmogorov complexity differs from data compression in that the difficulty of coding is disregarded, and is generally an uncomputable function. The decoding is only required to be computable, and not necessarily of low computational complexity. On the other hand, one advantage of Kolmogorov complexity is that it applies very generally. If one takes any recursive language which is of asymptotically low density, i.e., the fraction of strings in it approaches zero as the size grows, then it is easy to see that all but a finite number of its members have generating programs shorter by at least some amount. The program only needs to know the index of the string in the lexicographic ordering of the language to generate the string.

The main result of this paper is to show that if the density of a language goes to zero sufficiently quickly and the language is computable probabilistically in polynomial time, then there are probabilistic polynomial-time algorithms for coding and decoding

* Received by the editors November 9, 1987; accepted for publication (in revised form) August 24, 1990. A preliminary version of this paper appeared in the Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, Rhode Island, 1985.

† Department of Computer Science, Stanford University, Stanford, California 94305. This author's research was supported in part by National Science Foundation Presidential Young Investigator grant CCR-8858097. Part of this work was done while the author was at Computer Science Division, University of California, Berkeley, supported by a Fannie and John Hertz Foundation fellowship.

‡ Mathematics Department, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This author's research was supported by National Science Foundation grant NSF-89-12586 CCR and Air Force Office of Scientific Research grant AFOSR-89-0271.

which provide short representations for all but a finite number of its members. The encoding and decoding algorithms operate by transforming the language to one which may be compressed using arithmetic coding [El], [Ris], [RL]. One interesting property of our scheme is that the representation of a given string is not determined solely from the string itself but may also depend upon the random choices made by the encoding algorithm. The decoding algorithm also operates probabilistically, but without the knowledge of the random choices made by the encoding algorithm.

We consider this result to be of primarily theoretical significance. It is not directly relevant to practice because the algorithms are too slow to compete with algorithms presently in use, and they give compressions which are far from optimal in an information-theoretic sense. In general, we obtain a savings of only $O(\log n)$ bits on strings of length n. Sipser [S] shows how to obtain nearly optimal encodings under the same hypotheses but his encoding and decoding algorithms run in polynomial time only with an oracle for Σ_2^p.

One form of optimal compression is to represent a string in a language by its index in a lexicographic ordering of the language. We call this operation ranking. Even for polynomial-time languages it is not hard to show that ranking is generally #P hard to compute. We will show that one-way log space languages and context-free languages can be ranked in polynomial time. Results similar to our results on ranking were obtained independently by Allender [A] when he was studying classes of easily invertible functions.

We note that our results apply in the presence of any oracle. We give oracles under which our bounds cannot be significantly improved.

2. Definitions. In this section we define our notion of data compression on languages. There are two functions, an encoding or *compression* function, and a decoding or *decompression* function. We also allow for functions computed by probabilistic machines, where the function value depends also upon the random choices made by the machine. We call these *probabilistic functions* and consider them to be mappings from strings to random variables whose distribution is taken from the behavior of the machines.

NOTATION. Let $\Sigma=\{0, 1\}$; Σ^* be the collection of all strings; Σ^n, $\Sigma^{<n}$, and $\Sigma^{\leqq n}$ be the strings of length n, less than n, and less than or equal to n. For a language $L\subseteq\Sigma^*$ let L^i be the strings in L of length i. For a finite set S let $|S|$ be the cardinality of S. For a string s let $|s|$ be the length of s. The symbol λ denotes the empty string.

DEFINITION. A function $f:\Sigma^*\to\Sigma^*$ is a *compression function* of language L if f is one-to-one on L, and for all except finitely many $x\in L$, we have $|f(x)|<|x|$.

DEFINITION. A language L is *compressible* in time T if there are functions f and g where f is computable by a machine running in time $O(T(n))$ on inputs of length n, and g is computed by a machine running in time $O(T(n))$, where n is the length of the output; and for any $x\in L$, $g(f(x))=x$, the function f is a compression function for L.

DEFINITION. A function f *optimally compresses* a language L if for any $x\in L$ of length n,

$$|f(x)|\leqq\left\lceil \log\left(\sum_{i=0}^{n}|L^i|\right)\right\rceil.$$

A special kind of optimal compression is *ranking*. The ranking function r_L maps $x\in\Sigma^*$ to the number of strings in L that are less than or equal to x in a lexicographical ordering of Σ^*.

DEFINITION. A *probabilistic function* f is a mapping from Σ^* to random variables over Σ^*. If f, g, and h are probabilistic functions, we say $f \circ g = h$ if for each $x, y \in \Sigma^*$,

$$\Pr[h(x)=y] = \sum_{z \in \Sigma^*} \Pr[g(x)=z] \cdot \Pr[f(z)=y].$$

A standard function is considered to be a special case of a probabilistic function which always outputs a random variable that is constant.

DEFINITION. A probabilistic machine M *computes* the probabilistic function f if for all x,

$$\Pr[M \text{ on input } x \text{ halts with output } y] = \Pr[f(x)=y].$$

DEFINITION. A language L is *compressible* probabilistically in time T if there are probabilistic functions f and g such that f is computable by a probabilistic machine running in time $O(T(n))$ on inputs of length n, and g is computed by a probabilistic machine where each branch of the computation uses at most time $O(T(n))$, where n is the length of the output on that branch. For $|x| \to \infty$,

$$\Pr[g \circ f(x) = x] \to 1$$

and

$$\Pr[|f(x)| < |x|] \to 1.$$

3. Compressing polynomial-time languages.

DEFINITION. Given a language L, let $\mu_L : N \to R$ be defined by $\mu_L(n) = |L^n|/2^n$. In this section, we prove the following theorem.

THEOREM 1. *If $L \in P$, $k > 3$, and $\mu_L(n) \leqq n^{-k}$, then L can be compressed in probabilistic polynomial time. For each $x \in L$ of length n, the compression function yields strings of length $n - (k-3) \log n$*[1] *with probability approaching* 1.

Proof. Fix the input length n. Let $A = L^n$. For $s \in \Sigma^{\leqq n}$ let A_s be the strings in A that have prefix s. Let

$$\alpha_s = |A_s|/2^{n-|s|}$$

represent the density of A_s and for $|s| < n$, let

$$\begin{aligned}\beta_{s0} &= \alpha_{s0}/2\alpha_s \\ &= |A_{s0}|/|A_s| \quad (\text{let } 0/0 = 1),\end{aligned}$$

and

$$\begin{aligned}\beta_{s1} &= 1 - \beta_{s0} \\ &= |A_{s1}|/|A_s|.\end{aligned}$$

Thus β_{si} is the fraction of strings in A_s that have prefix si. Note that if $|s| = m$ and $s_0, s_1, \cdots, s_m$ are its $m+1$ prefixes, then $\alpha_s = \alpha_\lambda \cdot \beta_{s_0} \cdot \cdots \cdot \beta_{s_m}$. For any string s, let s' be the next higher string lexicographically, except that if s contains no 0's, we let s' be the special symbol ω. Recall that λ is the empty string. Let

$$\gamma_\lambda = 0$$

and

$$\gamma_\omega = 1.$$

[1] All logs are to the base 2.

For $s \in \Sigma^{<n}$ let

$$\gamma_{s0} = \gamma_s$$

and

$$\gamma_{s1} = \gamma_s + (\gamma_{s'} - \gamma_s)\beta_{s0}.$$

CLAIM 1. *If one assigns to each string s the subinterval* $[\gamma_s, \gamma_{s'})$ *then one obtains, for each* $m \leqq n$, *a partition of* $[0, 1)$ *among the strings of length m, where the length of the interval is proportional to the size of* A_s.

Proof. We may prove this by induction on m. The basis at $m = 0$ is clear since $[\gamma_\lambda, \gamma_\omega) = [0, 1)$.

To prove the induction step at m, we must check that for each s of length $m-1$ the interval $[\gamma_s, \gamma_{s'})$ is split by γ_{s1} so that it is proportionally divided between $s0$ and $s1$. This follows because β_{s0} gives the fraction of A_s that belongs to A_{s0}. This tells us the fraction of the interval $[\gamma_s, \gamma_{s'})$ that we must add to γ_s to give γ_{s1}. □

Let $t_1, t_2, \cdots, t_l$ be an enumeration of A in lexicographic order.

CLAIM 2. *The sequence* $\gamma_{t_1}, \gamma_{t_2}, \cdots, \gamma_{t_l}, \gamma_\omega$ *forms an arithmetic sequence beginning at* 0 *and ending at* 1.

Proof. The proof follows from Claim 1 since for every i, $|A_{t_i}| = 1$. Therefore the intervals $[\gamma_{t_i}, \gamma_{t_{i+1}})$ all have the same length. □

For each i let $h(t_i)$ be the first $\lceil \log |A| \rceil$ bits in the binary representation of γ_{t_i}. By the above claim h is a 1-1 function. Thus $h(t_i)$ may be seen as a unique representation for t_i. This idea is called *arithmetic coding* [El], [Ris], [RL]. The function h compresses the members of A optimally in an information-theoretic sense since it is clear that at least $\log |A|$ bits are necessary to uniquely specify members of A. However it is not hard to show that computing this coding is a #P-complete problem [V] and thus unlikely to be computable efficiently. Instead, we will show that there is a probabilistic polynomial-time machine F computing an approximation of h and a probabilistic polynomial-time machine G such that $G(F(x)) = 1$ with high probability.

The main idea is to obtain estimates to the values β_s by sampling, i.e., by choosing random extensions to s to find members of A_s. Two difficulties arise. One is that for some strings s, α_s may be very small because A_s is exponentially sparse. In that case we would be required to take exponentially many samples to find even one element in A_s. The second problem concerns the coordination between the compressor F and decompressor G. Both must compute approximations to the β's, but if their estimates differ by even a small amount it will be unlikely that the composition of the two machines will compute the identity function on A, as required.

We solve the first difficulty by recalling that $\alpha_\lambda = \mu_L(n) \leqq n^{-k}$ by the hypothesis of the theorem. Observe that for any $t \in A$, $\alpha_t = 1$, and that for any $s \in \Sigma^{<n}$, $\alpha_{s0}, \alpha_{s1} \leqq 2\alpha_s$. In other words, the densities associated with the prefixes of any given $t \in A$ start out small, end up at 1, and cannot increase by more than a factor of two at any point. Let u be the longest prefix of t such that $\alpha_u \leqq n^{-k}$. For all longer prefixes s, $\alpha_s > n^{-k}$ and therefore by taking polynomially many random extensions of s, one can obtain good estimates to α_s. We write t as uv and only apply the above arithmetic coding technique to v. The prefix u is left unencoded. We thus encode t by the string $ue(v)$ where $e(v)$ is the encoding of v. We also append to the front $\lceil \log n \rceil$ bits giving $|u|$, so that we can unambiguously parse this code.

We now solve the second difficulty by choosing the estimating values for the β's from a small set of standard approximations so that no two of them are nearly equidistant from any of the actual β's. Hence any two sampling runs will very likely obtain agreeing values for the β's. We select the estimating values as follows.

Choose any string $s \in A$ and let $s_0, s_1, \cdots, s_n$ be its $n+1$ prefixes. Consider the n values $\beta_{s_0 0}, \cdots, \beta_{s_{n-1}0}$. Let X be the set of numbers in $[0, 1)$ that are within $1/6n^2$ of any of these values, i.e.,

$$X = \left\{ x \in [0, 1) : |x - \beta_{s_i 0}| \leqq \frac{1}{6n^2} \text{ where } i \leqq n \right\}.$$

We view X as a collection of possibly overlapping intervals of length at most $1/3n^2$ each. Let $Y = \{i/n : 0 \leqq i \leqq n\}$ and, for $0 \leqq c \leqq n$, let $Y_c = Y + (c/2n^2)$, i.e., add the offset $c/2n^2$ to each value of Y.

CLAIM 3. *There is an integer c, such that $0 \leqq c \leqq n$ and $Y_c \cap X = \varnothing$.*

Proof. There are $n+1$ possible choices for c. Say a choice for c is bad if Y_c intersects one of the intervals in X. The points in Y are $1/n$ apart, the offsets are $1/2n^2$ apart, and the intervals are of length at most $1/3n^2$. Hence each interval in X may intersect at most one offset of one point in Y. So each interval in X may cause at most one of the choices for c to be bad. Since there are $n+1$ possible choices for c and at most n intervals, there remains a good choice for c satisfying the conditions of the claim. □

Fix a good c as above. Define $a : [0, 1) \to [0, 1)$ as follows. If $x \in Y_c$ then $a(x) = x$. If $x \notin Y_c$ but lies between two points y_1 and y_2 of Y_c then $a(x) = y_i$, where y_i is the nearest of the two to $\frac{1}{2}$, or the greater of the two if both are equidistant from $\frac{1}{2}$. If x is greater than or less than all members of Y_c then $a(x)$ is the nearest member of Y_c to x.

CLAIM 4. *If $\beta \in \{\beta_{s_0 0}, \cdots, \beta_{s_{n-1}0}\}$ and $|\tilde{\beta} - \beta| \leqq 1/6n^2$ then $a(\tilde{\beta}) = a(\beta)$.*

Proof. Since β and $\tilde{\beta}$ are in the same interval of X and no member of Y_c intersects X, the claim immediately follows. □

Let $\varepsilon = 3/n$.

CLAIM 5. *For any $\beta \in [0, 1)$*

(1) $a(\beta) \geqq (1-\varepsilon)\beta$;

(2) $1 - a(\beta) \geqq (1-\varepsilon)(1-\beta)$.

Proof. First prove 1.

If $\beta \in Y_c$ then $a(\beta) = \beta$.

If β lies between y_1 and y_2 in Y_c and if both y_1 and y_2 are less than or equal to $\frac{1}{2}$, then $a(\beta) > \beta$. If y_1 or y_2 are greater than $\frac{1}{2}$, then $\beta \geqq \frac{1}{2} - 1/n$ since $|y_1 - y_2| = 1/n$ and β lies between them. Additionally, $a(\beta) \geqq \beta - 1/n$ since $a(\beta)$ is y_1 or y_2. Therefore $a(\beta)/\beta \geqq (\beta - 1/n)/\beta = 1 - 1/n\beta$. Since $\beta \geqq \frac{1}{2} - 1/n$, it is easy to see that $1 - 1/n\beta \geqq 1 - 3/n$ for $n \geqq 6$. Thus $a(\beta) \geqq (1-\varepsilon)\beta$.

If β is greater than all members of Y_c then again $a(\beta) \geqq \beta - 1/n$ and $\beta \geqq \frac{1}{2} - 1/n$, so by the above argument $a(\beta) \geqq (1-\varepsilon)\beta$.

Finally, if β is less than all members of Y_c then $a(\beta) \geqq \beta$.

The proof of 2 follows similarly by an argument symmetric around the point $\frac{1}{2}$. □

CLAIM 6. *For any $s \in \Sigma^{<n}$, if R is a collection of n^{k+7} random extensions of s and we estimate β_{s0} using R then*

$$\Pr[|\textit{estimated value} - \beta_{s0}| > n^{-3}] < n^{-1}.$$

Proof. This is an application of Chebyshev's inequality (see, e.g., [F]). □

Let $m = n - |u|$ and let $B = \{s \in \Sigma^m : us \in A\}$. Redefine the γ's and h using B for A and m for n, i.e., $\gamma_\lambda = 0$, $\gamma_\omega = 1$, $\gamma_{s0} = \gamma_s$, and $\gamma_{s_1} = \gamma_s + (\gamma_{s'} - \gamma_s)\beta_{us0}$ for $s \in \Sigma^{<m}$. Also, for $t \in B$, $h(t) =$ the first $\lceil \log l \rceil$ bits of γ_t, where $l = |B|$. Thus h provides a coding for B, where if $t \in B$, $|h(t)| = \lceil \log |B| \rceil \leqq m - k\lfloor \log n \rfloor$ since $\alpha_u \leqq n^{-k}$ and thus $|B| \leqq 2^m \cdot \alpha_u \leqq m - \lfloor \log n^k \rfloor \leqq m - k\lfloor \log n \rfloor$.

For each $s \in \Sigma^{\leqq n}$ let $\bar{\beta}_s = a(\beta_s)$. Let $\bar{\gamma}_\lambda = 0$, $\bar{\gamma}_\omega = 1$, and for $s \in \Sigma^{<n}$ let $\bar{\gamma}_{s0} = \bar{\gamma}_s$ and $\bar{\gamma}_{s1} = \bar{\gamma}_s + (\bar{\gamma}_{s'}) - \bar{\gamma}_s \bar{\beta}_{s0}$.

CLAIM 7. *Let $s \in \Sigma^{\leqq m}$ and $j = |s|$. Then $(\bar{\gamma}_{s'} - \bar{\gamma}_s) \geqq (1-\varepsilon)^j(\gamma_{s'} - \gamma_s)$.*

Proof. The proof is by induction on j. The basis at 0 is immediate. To prove for j assuming that the claim holds for $j-1$, we first take the case where s is of the form $t0$.

Observe that

$$\bar{\gamma}_{(t0)'} = \bar{\gamma}_t + (\bar{\gamma}_{t'} - \bar{\gamma}_t)\bar{\beta}_{t0},$$
$$\bar{\gamma}_{t0} = \bar{\gamma}_t.$$

Thus

$$\begin{aligned}
\bar{\gamma}_{(t0)'} - \bar{\gamma}_{t0} &= (\bar{\gamma}_{(t')} - \bar{\gamma}_t)\bar{\beta}_{t0} \\
&\geqq (1-\varepsilon)^{j-1}(\gamma_{t'} - \gamma_t)\bar{\beta}_{t0} \quad \text{(by the induction hypothesis)} \\
&\geqq (1-\varepsilon)^{j-1}(\gamma_{t'} - \gamma_t)(1-\varepsilon)\beta_{t0} \quad \text{(by Claim 5)} \\
&= (1-\varepsilon)^j(\gamma_{t'} - \gamma_t)\beta_{t0} \\
&= (1-\varepsilon)^j(\gamma_{t1} - \gamma_{t0}).
\end{aligned}$$

The other case where s is of the form $t1$ is handled similarly, as follows. Observe that

$$\bar{\gamma}_{(t1)'} = \bar{\gamma}_{t'}$$
$$\bar{\gamma}_{t1} = \bar{\gamma}_t + (\bar{\gamma}_{t'} - \bar{\gamma}_t)\bar{\beta}_{t0}.$$

Thus

$$\begin{aligned}
(\bar{\gamma}_{(t1)'} - \bar{\gamma}_{t1}) &= \bar{\gamma}_{t'} - \bar{\gamma}_t - (\bar{\gamma}_{t'} - \bar{\gamma}_t)\bar{\beta}_{t0} \\
&= (\bar{\gamma}_{t'} - \bar{\gamma}_t)(1-\bar{\beta}_{t0}) \\
&\geqq (1-\varepsilon)^{j-1}(\gamma_{t'} - \gamma_t)(1-\bar{\beta}_{t0}) \quad \text{(by the induction hypothesis)} \\
&\geqq (1-\varepsilon)^{j-1}(\gamma_{t'} - \gamma_t)(1-\varepsilon)(1-\beta_{t0}) \quad \text{(by Claim 5)} \\
&= (1-\varepsilon)^j(\gamma_{t'} - \gamma_t)(1-\beta_{t0}) \\
&= (1-\varepsilon)^j(\gamma_{(t1)'} - \gamma_{t1}). \qquad \square
\end{aligned}$$

For $t \in B$ let $\bar{h}(t)$ equal the first $\lceil 21 + \log l \rceil$ bits in the binary representation of $\bar{\gamma}_t$. Since $\varepsilon = 3/n$, $\lim_{n\to\infty}(1/(1-\varepsilon)^n) = e^3 \leqq 21$. Hence for large n, $\bar{h}$ is a 1-1 function on B, by the above claim. This solves the second difficulty.

ENCODING ALGORITHM.

Input: $t \in A$, with prefixes $s_0, \cdots, s_n$. For each $i < n$ we take enough random extensions of s_i to determine with high probability whether α_{s_i} is significantly greater than n^{-k}. Let j be the maximum i such that the estimate of $\alpha_{s_i} < n^{-k}$, let $u = s_j$, and let $u\nu = t$. Estimate the β_{s_i0} for each $i \geqq j$ by taking n^{k+7} random extensions of s_i. Find a good c such that no $\beta_{s_i0} \in X$ as above. Obtain the values of $\bar{\beta}_{s_i0}$ and $\bar{\beta}_{s_i1}$. For $i \geqq j$ obtain $\bar{\gamma}_{s_i}$ from the $\bar{\beta}$'s and determine $\bar{h}(\nu)$.

For an integer m let *double*(m) be m written in binary with each bit repeated twice and terminated with 01. The final encoding of t, $f(t) = z_1z_2z_3z_4z_5$.

$$\begin{aligned}
z_1 &= \mathit{double}(n - |f(t)|) \\
z_2 &= j \\
z_3 &= u \\
z_4 &= c \\
z_5 &= \bar{h}(\nu).
\end{aligned}$$

DECODING ALGORITHM.

Given $f(t)$, obtain z_1, and compute n. Obtain z_2 and then z_3 and $z_4 = c$. This gives the set Y_c. By sampling we can determine with high probability $\bar{\beta}_{u0}$ and $\bar{\beta}_{u1}$ and hence $\bar{\gamma}_0$ and $\bar{\gamma}_1$. The first bit of ν is 0 if and only if $z_5 < \bar{\gamma}_1$. Continue in this way to get all of the bits of ν.

Analysis. We calculate $|f(t)|$. First $|z_1| \leqq 2 + \log\log n$ and $|z_2| = \lceil \log n \rceil$, and $|z_4| = 1 + \lceil \log n \rceil$. Also, $|z_3| = |u|$ and $|z_5| = |\nu| - \lceil k \log n \rceil + 1$. Hence $|f(t)| \leqq n - (k-3) \log n$ for large n. □

The following result, similar to Theorem 1, was pointed out to us by a referee.

Fact. If $L \in P$, k, $l > 0$, and $\mu_L(n) \leqq n^k/2^n$ then L can be compressed by $l \log n$ bits in deterministic polynomial time.

Note that L contains at most polynomially many strings of length n. A string of L of length n can be encoded by giving its prefix of length $n - (k+l) \log n$ followed by the index of the string among n-bit strings of L that have the same prefix.

4. Extensions and discussion. In the proof of Theorem 1, we have used the assumption that $L \in P$ to test for membership in L efficiently. Since the compression algorithm given in the proof is probabilistic, it is enough to assume that $L \in BPP$ to obtain the same result.

If L is very sparse, say $\mu_L \leqq 2^{-n/2}$, then Theorem 1 seems far from optimal because it gives only $O(\log n)$ bits of saving while one might expect to save $n/2$ bits. However, it is unlikely that one will be able to prove a significantly stronger result because we can construct an oracle for which our compression is close to the best possible.

THEOREM 2. *There is a language S such that $\mu_S = e^{-n}$ and S is not compressible by more than $O(\log n)$ bits by a probabilistic polynomial-time machine with an oracle for S.*

Proof. Let S be the language that, for each $n = 2^{2^m}$, contains exactly one string of length n which is Kolmogorov random, and no other strings. Note that there is a polynomial-time compression function for S given an oracle for S by mapping the strings in S^n to strings of length $\log |n|$. But this compression function has no efficient decompressor.

Let M be a probabilistic machine with an oracle for S which runs in time n^k. Let $s \in S$, and let t be a compression of s such that $|s| - |t| > (k+2) \log n + 1$. We show that M cannot restore s from t.

We will show that if M could restore s from t, then s would have a short description, which would contradict the Kolmogorov randomness of s. To show this, we show that we can describe in $(k+2) \log n + 2$ bits all information needed to answer the queries M could ask the oracle.

The language S is so sparse that on input of length $n = 2^{2^m}$, M has no time to write out a string in S of length greater than n. Therefore, the answers to all queries about strings longer than n are "no." Also, S is so sparse that we can write out all strings in S of length less than n in $2 \log n$ bits; this will enable us to answer all queries about strings of length less than n.

There is only one string of length n in S. We show how to give a short description of it. Consider the computation of M on t. By assumption, we know that $|t| < n - ((k+2) \log n + 2)$. For a given sequence of coin tosses, M produces a value depending on the oracle answers. There is exactly one string of length n in S, so without loss of generality we can assume that M never asks questions about strings of length n after it gets the "yes" answer on a question about a length n string. For a given coin

toss sequence, there can be at most n^k questions to the oracle and at most n^k+1 different sets of oracle answers on questions about strings of length n, depending on the position of the "yes" answer in the lexicographical ordering of the questions, including the possibility of no "yes" answers at all.

Information about the computation of M on t can be represented by a table with rows corresponding to coin toss sequences and columns corresponding to the n^k+1 possible positions of the "yes" answer. Table entries are the values M produces. Since M is a probabilistic machine, M must agree on at least one half of all values in the column that corresponds to the oracle for S. Let R be the number of rows and let C be the number of columns. C is at most n^k+1. It follows from the above that there is a group of at least $R/2$ table positions containing the same value. The total number of different values corresponding to the groups of at least $R/2$ table positions containing the same value is at most

$$\frac{RC}{R/2}=2C\leqq 2(n^k+1).$$

We can give an index of the value that corresponds to the right oracle using $k \log n+2$ bits.

The short description of s consists of three parts: all strings in S of length less than n (at most $2 \log n$ bits), the index of the table value ($k \log n+2$ bits), and t (less than $n-((k+2) \log n+2)$ bits). The total length of the description is less than n, and s can be reconstructed from the description—a contradiction. □

The above theorem shows that this compression is the best that can generally be achieved by a probabilistic polynomial-time algorithm with respect to an oracle. Perhaps the algorithm can be improved in a different way, by making it deterministic. We construct an oracle for a sparse language not compressible in deterministic polynomial time. This suggests that it would be difficult to eliminate the randomization in our compression algorithm.

THEOREM 3. *There is a language S where $\mu_S \leqq 2^{-n/2}$ and where S cannot be compressed by any deterministic polynomial-time machine using an oracle for S.*

Proof. The proof is by diagonalization. Let $M_1, M_2, \cdots$ be the list of all deterministic polynomial-time Turing machines with an oracle. Without loss of generality, we can assume that each M_i runs in time $n^{\log i}$. To each machine, we assign an infinite sequence of input lengths according to the sequence

$$112123123412345\cdots.$$

The nth element of the sequence is the index i of the machine we consider for input length n. We construct a set S, such that for every n, M_i with an oracle for S either does not map a string in S^n into a shorter string, or maps two distinct strings in S^n into the same string. Therefore, each machine M_i with an oracle for S fails to compress infinitely many elements of S, and the theorem follows if $\mu_S \leqq 2^{-n/2}$.

We construct S in steps. During step n, we simulate the machine M_i on all inputs of length n. At this step, the set S is already fixed for strings of length less than n, and if the machine asks the oracle questions about these strings, we answer "yes" or "no" depending on whether the string is in or out of S. We now determine which strings of length n to add to S. Whenever M_i asks an oracle a query about a string of length n, answer "yes." Assuming M_i gives an output on every string of length n, either there is a string x which M_i maps into a string of length at least n, or there are two distinct strings x_1 and x_2, which the machine maps into the same string.

In the first case, we add to S both x and all the strings of length n or more that M_i asked the oracle about when simulated with x as an input. This ensures that with the oracle for S, M_i works on x the same way as during the simulation, and therefore does not compress x.

In the second case, we add to S both x_1 and x_2, and all the strings of length n or more that M_i asks about when simulated on inputs x_1 and x_2. This ensures that with the oracle for S, M_i maps x_1 and x_2 into the same string.

It remains to show that $\mu_S \leqq 2^{-n/2}$. The set S^n is fixed after the nth step. The number of elements in S^n does not exceed 2+(the number of questions asked during steps 1 through n). Since the running time of M_i is $n^{\log i}$,

$$|S^n| \leqq 2 + n \cdot n^{\log n} < 2^{n/2}$$

for n large enough. □

5. Incompressible languages.

THEOREM 4 (Levin [L]). *Assume that there is a one-way function. Let R be the set of pseudorandom sequences generated by a cryptographically strong pseudorandom bit generator* [BM], [Y]. *Then R cannot be compressed by any function computable in probabilistic polynomial time.*

Thus, if a one-way function exists, we cannot strengthen Theorem 1 to apply to all languages in NP.

Proof. If f compresses R in probabilistic polynomial time, then the following statistical test, S, distinguishes the members of R from truly random sequences [Y]. Let $S(x)=1$ if and only if $x = M_2(M_1(x))$, where M_1 computes f and M_2 computes f^{-1}. Thus, $S(x)=1$ for at most half the sequences of length less than or equal to n, but for all the sequences in R. □

The next theorem shows how to construct a language incompressible probabilistically in polynomial time without assuming the existence of a one-way function. The result is not stronger than the above, however, because the language we construct is in double exponential time.

THEOREM 5.

(A) *There is an exponential-time language that cannot be compressed in deterministic polynomial time.*

(B) *There is a double exponential-time language that cannot be compressed in probabilistic polynomial time.*

Proof. The proof is similar to the proof of Theorem 3. First, we construct a language, S, which is not compressible by any polynomial-time Turing machine.

Consider a machine computing a function $g: \Sigma^* \to \Sigma^*$ on inputs of length n. Since the number of strings of length n is greater than the number of strings of length less than n, one of the following two statements must be true:

(1) there is a string x such that $|g(x)| \geqq |x|$, or

(2) there are two strings, x_1 and x_2, such that $g(x_1) = g(x_2)$.

In the first case we let b be the lexicographically smallest such string, and make $S^n = \{b\}$. In the second case we let b_1, b_2 be the lexicographically smallest pair of such strings, and make $S^n = \{b_1, b_2\}$. This ensures that g is not a compression function for S^n. It remains to describe for which input length each polynomial-time machine will be tricked as above.

Consider a list $M_1, M_2, \cdots$ of all Turing machines. To each machine M_i on the list, we add an n^i timer to construct a new machine M_i' such that if the time runs out before M_i halts, M_i' rejects; otherwise, M_i' simulates M_i. It is easy to see that each polynomial-time function is computed by M_i' for some i.

To each machine M_i we assign infinitely many input lengths for which it will be tricked. The strings are assigned according to the sequence

$$112123123412345 \cdots.$$

The nth element of the sequence corresponds to the index of the machine to be tricked on the inputs of length n.

The language S described above has the property that for any polynomial-time computable function f, there are infinitely many input lengths n such that f is not a compression function for S^n. To prove part (A) of the theorem, it remains to show that language S is in exponential time.

Given a string x of length n, we find the machine M_i' which corresponds to n. We simulate the machine on inputs of length n to find b or b_1 and b_2 described above, and check if x is in S. Since M_i' runs in time $n^i \leqq n^n$ and the bottleneck is the simulation of M_i' on all inputs of length n, it is easy to see that the process takes exponential time.

To prove part (B) of the theorem, we consider all probabilistic polynomial-time machines M_i and proceed as above. The complexity of the resulting language is double exponential since it takes an exponential amount of time to simulate a probabilistic polynomial-time machine. □

6. Ranking. Recall that the ranking function for a language L, r_L, maps its input x into the number of strings in L that are less than or equal to x in a lexicographical ordering of Σ^*. In this section we will study languages for which r_L is computable in polynomial time. Note that if $r_L: L \to N$ is computable in polynomial time, then, using binary search, $r_L^{-1}: N \to L$ is computable in time polynomial in the length of its output. If L is sparse, then r_L is a compression function for L. In fact, r_L is an optimal compression function.

6.1. Easy-to-rank languages. The next two theorems show that languages in two natural language classes can be ranked in deterministic polynomial time.

THEOREM 6. *If a language L is accepted by a deterministic one-way log space Turing machine, then r_L can be computed in polynomial time.*

Proof. Let M be a one-way log space Turing machine that accepts L. Given a string x, we want to compute $r_L(x)$. We modify M to accept exactly the set $L \cap LE(x)$, where $LE(x)$ is a set of all strings lexicographically less than x. The modified machine, M_x, is constructed as follows.

First, we construct a finite automata A that compares its input with x, and accepts if and only if the input does not exceed x. M_x runs A and M in parallel, accepting if both machines accept, and rejecting otherwise. It is easy to see that M_x can be constructed in time polynomial in the length of x.

Next, we construct a graph $G=(V, E)$ such that vertices in V correspond to configurations of M_x on inputs of length not exceeding the length of x, and edges correspond to moves of M_x (by configuration we mean here a triple (*state*, *input tape position*, *work tape content*)). Then there is a one-to-one correspondence between the accepting computations of m_x and the set of paths from the initial to the final configurations in G (i.e., paths from the node that corresponds to the initial configuration of m_x to the node that corresponds to a final configuration). Consequently, there is a one-to-one correspondence between the paths and the members of $L \cap LE(x)$.

Notice that G is loop-free because M_x runs in bounded (polynomial) time. We can use dynamic programming to count the number of initial-to-final state paths in G in time polynomial in the size of G. The size of G is polynomial in x, because M_x is

a log space machine of size polynomial in $|x|$. Therefore we can compute in polynomial time the cardinality of $L \cap LE(x)$, which is equal to $r_L(x)$. □

Remark. The above proof can be easily modified to show that, given a finite automata, the size of the language accepted by the finite automata can be found in polynomial time. This is because either the automata accepts an infinite language (which can be determined in polynomial time), or every initial-to-final state path in its graph is simple, and we can use dynamic programming as above.

THEOREM 7. *If L is an unambiguous* CFL *given by an unambiguous grammar, then r_L can be computed in polynomial time.*

Proof. Without loss of generality, we can assume that the grammar for L is in Chomsky Normal Form and lambda free since, if we can compute $r_{L\{\lambda\}}$, we can easily compute r_L. Given a string x, we want to compute $r_L(x)$. Again, our main tool will be dynamic programming.

First, we show that we can compute in polynomial time the function $\#(A, n)$ which, for a nonterminal A of the grammar and an integer n, returns the number of strings of length n derivable from A. This function can be computed from the following equations:

$$\#(A, 1) = |\{A \to a : a \in \Sigma^*\}| \tag{5.1}$$

and

$$\#(A, n) = \sum_{A \to BC} \sum_{i=1}^{n-1} \#(B, i)\,\#(C, n-i). \tag{5.2}$$

The equation counts the number of derivations of length n strings from A. Since the grammar is unambiguous, the number of length n strings derivable from A is the same as the number of derivations.

To compute $\#(A, n)$, we use (5.1) and (5.2) to compute $\#(X, j)$ for all nonterminals X and all j less than n, and then compute the desired number. Since the size of the grammar is fixed (i.e., the number of nonterminals and productions is constant), the time needed to compute $\#(A, n)$ is linear in $n = |x|$.

Since context-free language membership can be tested efficiently, it is enough to show that the function $l(A, x)$, which equals the number of strings strictly less than x derivable from A, can be computed efficiently.

If x is a null string, then

$$l(A, x) = 0. \tag{5.3}$$

If $|x| = n \geqq 1$, then

$$\begin{aligned} l(A, x) = \sum_{i=0}^{n-1} \#(A, x) + \sum_{i=1}^{n-1} \sum_{A \to BC} [\,&l(B, x(1, i))\#(C, n-i) \\ &+ \sigma(B, x(1, i))\,l(C, x(i+1, n))]. \end{aligned} \tag{5.4}$$

In (5.4) $x[i]$ refers to the ith symbol of x, $x(1, i-1)$ refers to the substring of x formed by the first $i-1$ characters, and $\sigma(S, w)$ is the characteristic function that is 1 if and only if w is derivable from S. Equation (5.4) states that a string is less than x if it is shorter than x, or if it is of the same length and either has a prefix which is less than the corresponding prefix of x or the prefix is the same but the remaining portion of the string is less than the remaining portion of x. There is no double counting because L is unambiguous.

Since σ can be computed in cubic time, it is easy to see that l and therefore r_L can be computed in polynomial time. □

Note that if L can be parsed in linear time, then the above algorithm computes r_L in n^2 time.

Remark. The result of Theorem 7 has been strengthened in [BGS], where it is shown that unambiguous CFLs can be ranked in NC^2.

6.2. Approximating the ranking function. Assume L is a finitely ambiguous CFL, i.e., the number of derivations of every string in L is bounded by some constant k. We map every string x in the language into the number of derivations of strings less than or equal to x, denoted by $c(x)$. This can be computed efficiently using the algorithm from the proof of the previous theorem. Because the language is finitely ambiguous, we know that $c(x) \leqq kr_L(x)$, and

$$\log(c(x)) - \log(r_L(x)) \leqq \lceil \log(k) \rceil.$$

It follows that c is a good compression function for L; it is within a constant number of bits of the optimal compression achieved by the ranking function.

In general, if we can compute a monotone approximation $g(x)$ of $r_L(x)$, which is within a factor of $h(|x|)$, this approximation comes within $\lceil \log(h|x|) \rceil$ bits from the optimal compression achieved by the ranking function, and is a compression if L is sparse enough. This is interesting because, as we will see, computing the ranking function exactly is NP-hard for many languages; however it could be that approximating the ranking function is not so hard.

6.3. Hard-to-rank languages. In this section we show that there are some languages of a relatively low complexity (like 2-pebble languages) which are hard to rank.

Let SAT be the problem of testing the satisfiability of boolean formulas. Consider the set of tuples (f, a), where f is a SAT instance and a is an assignment of variables. Let L be the subset of tuples for which a is a satisfying assignment. It is easy to see that membership in L can be checked by a (two-way) log space Turing machine; in fact, it can be checked by a 2-pebble finite automaton. However, the following theorem shows that L cannot be ranked unless $P = \#P$ (which implies $P = NP$).

THEOREM 8 (Blum [B]). *If r_L can be computed in polynomial time, then the number of satisfying assignments to an instance of* SAT *can be computed in polynomial time.*

Proof. Let f be a boolean formula for which we want to compute $\#(f)$, the number of satisfying assignments. Let $Pred(f)$ be the biggest instance of SAT that is smaller than f. Let a_1 and a_2 be the biggest assignment of variables in f and $Pred(f)$, respectively. Then $\#(f) = r_L(f, a_1) - r_L(Pred(f), a_2)$ by the definition of ranking.

It follows that if r_L can be computed in polynomial time, $\#(f)$ can be computed in polynomial time.

Acknowledgment. We thank Paul Feldman for pointing out an error in an earlier draft of this paper.

REFERENCES

[A] E. W. ALLENDER, *Invertible functions*, Ph.D. thesis, Computer Science Department, Georgia Institute of Technology, Atlanta, GA, 1985.

[B] M. BLUM, personal communication.

[BGS] A. BERTONI, M. GOLDWURM, AND N. SABADINI, *Computing the counting function of context-free languages*, in Proc. 4th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 247, Springer-Verlag, Berlin, New York, 1987, pp. 169–179.

[BM] M. BLUM AND S. MICALI, *How to generate cryptographically strong sequences of random bits*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, Chicago, IL, 1982, pp. 112–117.

[El] P. ELIAS, unpublished.

[F] W. FELLER, *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1971.

[K] A. N. KOLMOGOROV, *Three approaches to the definition of information*, Probl. Pederachi Inform. 1 (1965), pp. 3-11.

[L] L. A. LEVIN, personal communication.

[Ris] J. J. RISSANEN, *Generalized Kraft inequality and arithmetic coding*, IBM J. Res. and Devel., 20 (1976), pp. 198-203.

[RL] J. J. RISSANEN AND G. G. LANGDON, JR., *Arithmetic coding*, IBM. J. Res. and Devel., 23 (1979), pp. 149-162.

[S] M. SIPSER, *A complexity theoretic approach to randomness*, in Proc. 15th ACM Symposium on Theory of Computing, (1983), pp. 330-335.

[V] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.

[Y] A. C. YAO, *Theory and applications of trapdoor functions*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, Chicago, IL, 1982, pp. 80-91.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 537-552, June 1991

© 1991 Society for Industrial and Applied Mathematics
010

ALGORITHMS FOR SCHEDULING IMPRECISE COMPUTATIONS WITH TIMING CONSTRAINTS*

WEI-KUAN SHIH†, JANE W. S. LIU†, AND JEN-YAO CHUNG‡

Abstract. Here the problem of scheduling tasks, each of which is logically decomposed into a mandatory subtask and an optional subtask, is considered. The mandatory subtask must be executed to completion in order to produce an acceptable result. The optional subtask begins after the mandatory subtask is completed and refines the result in order to reduce the error in the result. The optional subtask can be left incomplete. The error in the result of a task is equal to the processing time of the unfinished portion of the optional subtask. Two preemptive algorithms for scheduling, on a uniprocessor system, n dependent tasks with rational ready times, deadlines, and processing times are described. An algorithm is optimal in the following sense: whenever feasible schedules that meet the ready time and deadline constraints of all tasks exist, it finds one that has the minimum total error of all tasks. One of the algorithms is optimal when the tasks have identical weights, and its time complexity is $O(n \log n)$. The other algorithm has time complexity $O(n^2)$, but is optimal when tasks have different weights. A schedule is said to satisfy the 0/1 constraint when every optional subtask is either completed or discarded. The problem of finding an optimal feasible schedule that satisfies the 0/1 constraints and minimizes the total processing time of the discarded optional subtasks is NP-complete. Two algorithms for finding optimal schedules of dependent tasks on a uniprocessor system for the special case when all optional subtasks have identical processing times are presented.

Key words. real-time systems, scheduling to meet deadlines, deterministic scheduling

AMS(MOS) subject classification. 68Q25

1. Introduction. In a hard real-time system, it is essential for every (real-time) task to meet its timing constraint, that is, its execution begins after its ready time and completes by its deadline. Otherwise, a timing fault is said to occur, and the result produced by the task is of little or no value. Unfortunately, it is not always feasible to schedule all tasks to meet their timing constraints. An approach to avoidance of timing faults is to trade off the quality of the results produced by the tasks with the amounts of processor time required to produce them. Such a tradeoff can be realized by using the imprecise computation technique [1]-[4]. In a system that supports imprecise computations, a task can be logically decomposed into a mandatory subtask and an optional subtask. The mandatory subtask is the portion of the computation that must be done in order to produce a usable result of acceptable quality. This subtask must be completed before the deadline of the task. The optional subtask is the portion of the computation that begins after the mandatory subtask is completed and refines the result produced by the mandatory subtask. The quality of the intermediate result produced by an optional subtask is nondecreasing as it executes longer. If it is allowed to execute until completion by its deadline, the result produced by it is the desired, precise one. When it is not feasible for an optional subtask to complete by its deadline, however, the subtask is terminated at its deadline, producing an approximate result.

We are concerned with the problem of imprecise scheduling: each of the n preemptable tasks to be scheduled (1) consists of a mandatory subtask and an optional

* Received by the editors November 27, 1989; accepted for publication (in revised form) June 6, 1990. This work was partially supported by U.S. Navy Office of Naval Research contracts NVY N00014 87-K-0827 and NVY N00014 89-J-1181.

† Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801.

‡ IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

subtask and (2) has rational ready time, deadline, and processing time. A scheduling algorithm is *optimal* in the following sense. It determines whether feasible schedules that meet timing constraints of all tasks exist, and when such schedules exist it finds one that minimizes the total processing time of the unfinished portions of optional subtasks. This scheduling problem can be formulated as a network flow problem in the cases of dependent tasks on a uniprocessor system and independent tasks on an identical, multiprocessor system. Optimal algorithms based on this formulation have time complexity $O(n^2 \log^2 n)$ when the tasks have identical weights and $O(n^6)$ when the tasks have different weights [5]. (Weights of tasks measure their relative importance.) The special case in which all tasks are optional is considered in [6]. In this paper, we present two preemptive algorithms. These algorithms have time complexity $O(n \log n)$ and $O(n^2)$; they find optimal schedules of dependent tasks on a uniprocessor system when the tasks have identical weights and different weights, respectively.

Some applications may require that every task is executed satisfying the 0/1 constraint. We say that the execution of a task satisfies the 0/1 *constraint* if its optional subtask is either completed before its deadline or is not scheduled for execution, that is, is *discarded* entirely. The problem of scheduling tasks with primary and alternate versions [7] can also be formulated as one of scheduling with 0/1 constraint. In this formulation, the alternate version, with shorter processing time, is modeled as a mandatory subtask. The primary version is modeled as a mandatory subtask, with processing time equal to that of the alternate version, and an optional subtask, with processing time equal to the difference between the processing times of the primary version and the alternate version. The latter must be either completed or discarded entirely. We say that a schedule satisfies the 0/1 constraint when the execution of every task according to the schedule satisfies the 0/1 constraint. We show here that the problem of finding optimal schedules satisfying the 0/1 constraint, meeting timing constraints and minimizing the total processing time of the discarded optional subtasks, is NP-complete even when the tasks have identical weights. The special case where tasks have identical weights and all optional subtasks have equal processing time can be solved in polynomial time. We present two algorithms that find optimal schedules of dependent tasks on uniprocessor systems in the cases of arbitrary ready times and equal ready times. Their complexities are $O(n^2)$ and $O(n \log n)$, respectively.

The algorithms described in this paper complement the heuristic algorithms described in [1]-[3] for preemptive scheduling of independent, periodic jobs on uniprocessor systems to minimize total error. Our work also complements the queueing theoretical results on task scheduling to optimally trade off between average response time and result quality on uniprocessor systems [8]-[9].

The remaining part of this paper is organized as follows. Section 2 discusses the basic workload model used to characterize imprecise computations and the performance criterion used in this study. Two basic algorithms for scheduling dependent tasks on uniprocessor systems to meet deadlines and minimize the total processing time of the discarded portions of all optional subtasks are described in §§ 3 and 4. Section 5 shows that the problem of scheduling with 0/1 constraint and timing constraints to minimize the total processing time of discarded optional subtasks is NP-complete and presents two efficient algorithms for finding optimal schedules for the special case when all tasks have the same weight and the optional subtasks have the same processing time. Section 6 summarizes our results and discusses future work.

2. The basic imprecise scheduling problem. The problem of imprecise scheduling can be formulated as follows. We are given a set of preemptable tasks $\mathbf{T} =$

$\{T_1, T_2, \cdots, T_n\}$ in which each task T_i is characterized by the following parameters, which are rational numbers:

(1) *ready time* r'_i at which T_i becomes ready for execution,

(2) *deadline* d'_i by which T_i must be completed,

(3) *processing time* τ_i, which is the time required to execute T_i to completion in the traditional sense, and

(4) *weight* w_i, which is a positive number and measures the relative importance of the task.

Logically, each task T_i is decomposed into two subtasks: the *mandatory* subtask M_i and the *optional* subtask O_i. Hereafter, we refer to M_i and O_i simply as tasks. We use M_i and O_i to mean specifically the mandatory task and the optional task of T_i, respectively, and use T_i to mean the task as a whole. Let m_i and o_i be the processing times of M_i and O_i, respectively, m_i and o_i are rational numbers, and $m_i + o_i = \tau_i$. The ready time and deadline of the tasks M_i and O_i are the same as that of T_i.

A *schedule* on a uniprocessor system is an assignment of the processor to the tasks in $\mathbf{T}$ in disjoint intervals of time. A task is said to be scheduled in a time interval if the processor is assigned to the task in the interval. In any valid schedule, the processor is assigned to only one task at any time, and every task T_i is scheduled after its ready time. Moreover, the total length of the intervals in which the processor is assigned to T_i, referred to as the total *amount of processor time* assigned to the task, is at least equal to m_i and at most equal to τ_i. A task is said to be *completed in the traditional sense* at an instant t when the total amount of processor time assigned to it between its ready time and t becomes equal to its processing time. A mandatory task M_i is said to be completed when it is completed in the traditional sense. The optional task O_i is dependent on the mandatory task M_i; it becomes ready for execution when M_i is completed. O_i may be terminated at any time, however, even if it is not completed at the time; no more processor time is assigned to it after it is terminated. A task T_i is said to be completed in a schedule whenever its mandatory task is completed. It is said to be terminated when its optional task is terminated. The traditional workload model is a special case of our model in which $o_i = 0$ for all i.

The dependencies between the tasks in $\mathbf{T}$ are specified by their precedence constraints; they are given by a partial order relation $<$ defined over $\mathbf{T}$. $T_i < T_j$ if the execution of T_j cannot begin until the task T_i is completed and terminated. T_j is a *successor* of T_i if $T_i < T_j$. In order for a schedule of $\mathbf{T}$ to be valid, the precedence constraints between all tasks must be satisfied. A set of tasks is said to be independent if the partial order relation $<$ is empty, that is, the tasks can be executed in any order.

A valid schedule is a *feasible* one if in which every task is completed by its deadline. It is possible that the deadline of a task is later than that of its successors. Rather than working with the given deadlines, we use the modified deadlines, which are consistent with the precedence constraints and are computed as follows. The modified deadline d_i of a task T_i that has no successors is equal to its given deadline d'_i. Let $\mathbf{A}_j$ be the set of all the successors of T_j. The modified deadline d_j of T_j is $\min\{d'_j, \min_{T_k \in \mathbf{A}_j}\{d_k\}\}$. Similarly, the given ready time of a task may be earlier than that of its predecessors. We modify the ready times of tasks as follows. The modified ready time r_i of a task T_i that has no predecessors is equal to its given ready time r'_i. Let $\mathbf{B}_j$ be the set of all the predecessors of T_j. The modified ready time r_j of T_j is

$$\max\left\{r'_j, \max_{T_k \in \mathbf{B}_j}\{r_k\}\right\}.$$

It has been shown in [10] that a feasible schedule on a uniprocessor system exists for

a set $\mathbf{T}$ of tasks with the given ready times and deadlines if and only if there exists a feasible schedule of $\mathbf{T}$ with the modified ready times and deadlines. Working with the modified ready times and deadlines allows the precedence constraints to be ignored temporarily. If an algorithm finds an invalid schedule in which T_i is assigned a time interval after some time intervals assigned to T_j but $T_i < T_j$, a valid schedule can be constructed by exchanging the time intervals assigned to T_i and T_j to satisfy their precedence constraint without violating their timing constraints. In our subsequent discussion, by ready time and deadlines, we mean modified ready times and deadlines.

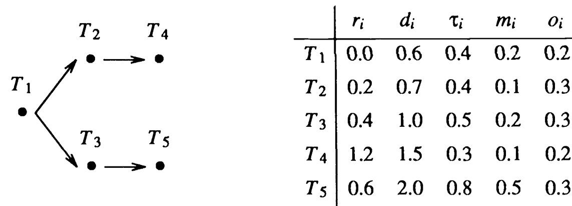
In all the schedules considered here, an optional task O_i is executed only in the time interval between its ready time and deadline; no processor time is assigned to it after its deadline d_i. When the amount of processor time σ_i assigned to O_i in a schedule is equal to o_i, we say that the task O_i and, hence, the task T_i are *precisely* scheduled. The *error* ε_i in the result produced by T_i (or simply the error of T_i) is zero. Otherwise, if σ_i is less than o_i, the error of T_i is equal to

$$\varepsilon_i = o_i - \sigma_i. \tag{1a}$$

In this case, we say that a portion of O_i of length $o_i - \sigma_i$ is discarded in the schedule. For a given schedule, the *total error* of the task set $\mathbf{T}$ is

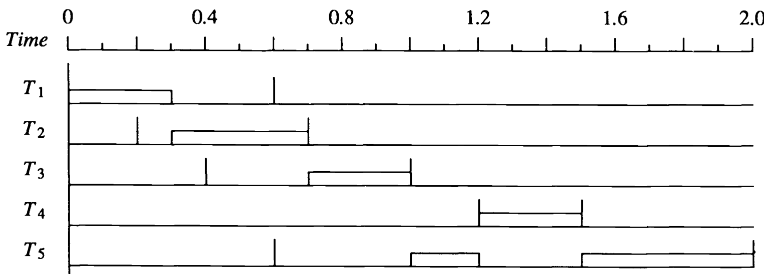
$$\varepsilon = \sum_{i=1}^{n} w_i \varepsilon_i. \tag{1b}$$

Again, $w_i > 0$ are the weights of the tasks. A schedule is said to be *precise* if the total error ε of the task set executed according to the schedule is zero. Only precise schedules are valid schedules in the traditional sense. As an illustrative example, we consider the task set shown in Fig. 1(a). In this directed graph, there is an edge from T_i to T_j



	r_i	d_i	τ_i	m_i	o_i
T_1	0.0	0.6	0.4	0.2	0.2
T_2	0.2	0.7	0.4	0.1	0.3
T_3	0.4	1.0	0.5	0.2	0.3
T_4	1.2	1.5	0.3	0.1	0.2
T_5	0.6	2.0	0.8	0.5	0.3

(a)



(b)

FIG. 1. *An example of imprecise schedules.*

if $T_i < T_j$ and there is no task T_k which is such that $T_i < T_k < T_j$. This task set cannot be feasibly scheduled if every task must be precisely scheduled. However, a feasible schedule with total error $\varepsilon = 0.4$ exists, and it is shown in Fig. 1(b).

The imprecise scheduling problem is as follows. Given a set of tasks $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$, we want to find an optimal schedule that is feasible and has minimum total error as defined in (1). Our algorithms make use of a modified version of the classical, earliest-deadline-first algorithm [10]. Like the classical algorithm, our version is also preemptive and priority-driven. The priorities of tasks are assigned according to their deadlines; tasks with earlier deadlines have higher priorities. (A tie between any two tasks with the same deadline is broken by letting the task with the smaller index have the higher priority.) However, according to our version of this algorithm, every task is terminated at its deadline even if it is not completed at the time; no processor time is assigned to any task in the time interval after its deadline. Scheduling decisions are made at the ready time and deadline of every task and at every instant when a task is completed in the traditional sense. At every instant, the processor is assigned to the task with the highest priority among the tasks that are ready to be executed, preempting any task with a lower priority if necessary. We refer to this algorithm as the ED *Algorithm.* The complexity of this algorithm is $O(n \log n)$. This algorithm always finds a schedule in which every task is scheduled in the time interval between its ready time and deadline. We will use this algorithm to determine whether a task set **T** can be feasibly scheduled, that is, whether every mandatory task completes before its deadline. The feasibility test is done by using the ED Algorithm to schedule the mandatory set $\mathbf{M} = \{M_1, M_2, \cdots, M_n\}$ alone. If the resultant schedule of **M** is precise, then the task set **T** can be feasibly scheduled. Otherwise, no feasible schedule of **T** exists [10].

3. Scheduling tasks with identical weights to minimize total error. The ED Algorithm can be used to find feasible schedules with minimum total error of any task set in which all tasks are optional and have identical weights.

THEOREM 1. *The* ED *Algorithm is optimal if the processing times of all mandatory tasks in* $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$ *are zero, and the tasks have identical weights.*

Proof. That the schedule S obtained using the ED Algorithm is feasible is obvious. We need only to show that the total error of the task set **T** is minimum when the tasks are executed according to S. Clearly, if the processor never idles between the earliest ready time a and the latest deadline of all tasks in **T**, the total error is minimum.

Suppose that the schedule S is as shown in Fig. 2(a); in S, the first idle period begins at t_0 and ends at t_1. Let $T_1, T_2, \cdots, T_j$ denote the tasks that are scheduled in the interval $[a, t_0]$ and $T_{j+1}, T_{j+2}, \cdots, T_n$ denote the tasks that are scheduled after t_1. Since the processor is never left idle intentionally, t_1 must be the earliest ready time of the tasks $T_{j+1}, T_{j+2}, \cdots, T_n$; the processor is left idle in the interval $[t_0, t_1]$ because these tasks cannot be scheduled earlier. We, therefore, can consider the segment $[a, t_0]$ in S independently of the later segments. The total error of **T** is minimized if the total error of the tasks scheduled in every such independent segment of S is minimized, that is, every segment of S is optimal.

To show that the total error of $T_1, T_2, \cdots, T_j$ is minimized in S, let T_j be the task with the latest finishing time among these tasks. We need to consider two cases: either t_0 is the deadline of T_j, or T_j completes in the traditional sense, with zero error, at t_0. In the former case, the segment $[a, t_0]$ of S is optimal since t_0 is the latest among all the deadlines $d_1, d_2, \cdots, d_j$ $(=t_0)$ and the processor never idles between a and t_0. Therefore, only the latter case needs to be considered further.

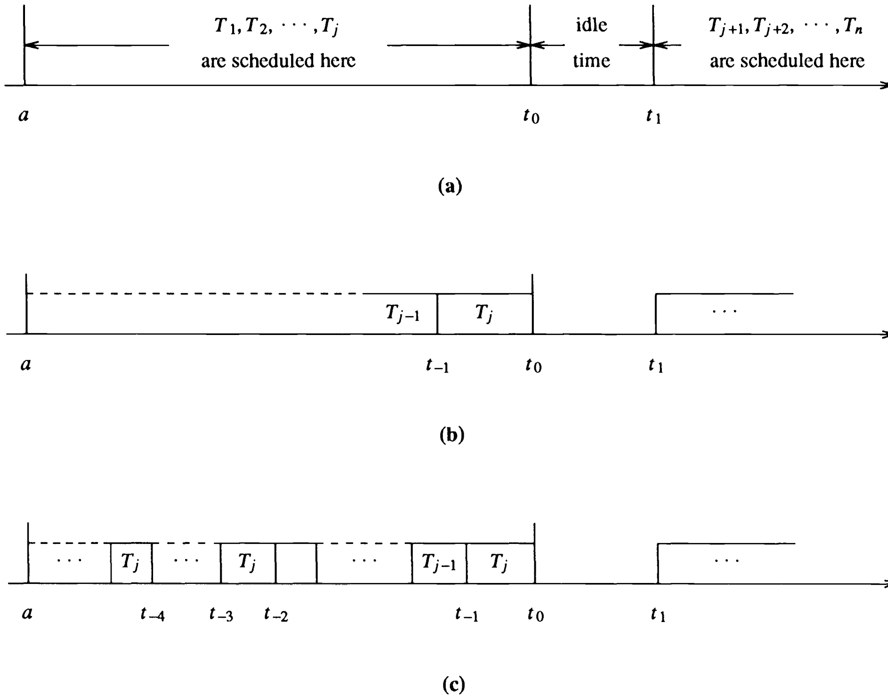


FIG. 2. *A possible* ED *schedule.*

We now show by induction that $T_1, T_2, \cdots, T_j$ are optimally scheduled with minimum total error if T_j completes in the traditional sense at t_0 and ε_j is equal to zero. This statement is clearly true if $j = 1$. Suppose that any segment in $[a, t_0]$ containing less than j tasks is optimal. If T_j is assigned a contiguous interval of time as shown in Fig. 2(b), the segment $[a, t_0]$ is optimal since the error of T_j is zero. Alternatively, T_j may be assigned noncontiguous intervals of time as shown in Fig. 2(c). In this case, the tasks scheduled between adjacent intervals that are assigned to T_j cannot be scheduled earlier. For example, the tasks scheduled in the interval $[t_{-2}, t_{-1}]$ in Fig. 2(c) must have ready time equal to or later than t_{-2}. Moreover, either t_{-1} is the deadline of T_{j-1}, or T_{j-1} completes in the traditional sense at t_{-1}. In either case, the total error of these tasks is minimized in S. Similarly, the total error of the tasks scheduled in $[t_{-4}, t_{-3}]$ is minimized, and so on. It follows that the segment $[a, t_0]$ is optimal. Similarly, we can show that the other independent segments of S are optimal. □

Our $O(n \log n)$ algorithm, called *Algorithm* F, for optimally scheduling a set of n tasks with identical weights on a uniprocessor system works as follows. Again, we are given a task set $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$ that is decomposed into two sets, the set of mandatory tasks $\mathbf{M} = \{M_1, M_2, \cdots, M_n\}$ and the set of optional tasks $\mathbf{O} = \{O_1, O_2, \cdots, O_n\}$. Algorithm F consists of three steps:

ALGORITHM F.

Step 1. Treat all mandatory tasks in $\mathbf{M}$ as optional tasks. Use the ED Algorithm to find a schedule S_t of the set $\mathbf{T}$. If $\mathbf{T}$ is a precise schedule, stop. The resultant schedule has zero error and is, therefore, optimal. Otherwise, carry out Step 2.

Step 2. Use the ED Algorithm to find a schedule S_m of the set **M**. If S_m is not a precise schedule, **T** cannot be feasibly scheduled. Stop. Otherwise, carry out Step 3.

Step 3. Transform S_t into an optimal schedule that is feasible and minimizes the total error using S_m as a template.

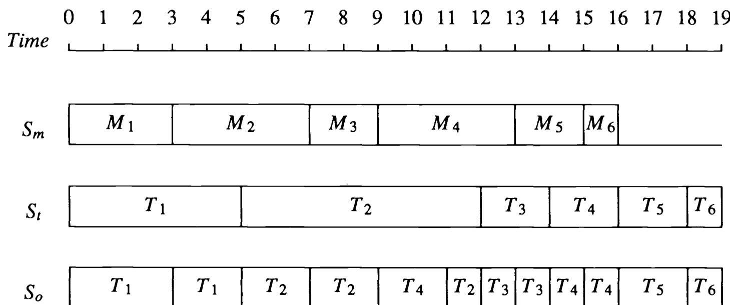
An illustrative example is shown in Fig. 3. We note that the schedule S_t obtained in Step 1 is not a valid one. Specifically, T_4 is assigned only two units of processor time, which is less than m_4. Using S_m as a template, we adjust the amounts of processor time assigned to the different tasks in S_t to obtain the optimal schedule S_o shown in Fig. 3(b).

3.1. Processor time adjustment process. The (processor time) adjustment process in Step 3 has as inputs the schedules S_m and S_t. The following information is needed:

(1) Time intervals $[a_j, a_{j+1}]$ for $j=1, 2, \cdots, k$. Let a_1 be the earliest starting time and a_{k+1} be the latest finishing time of all tasks in the schedule S_m. We partition the time interval $[a_1, a_{k+1}]$ according to S_m into disjoint intervals such that in S_m the processor is assigned to only one task in each of these intervals and is assigned to different tasks in adjacent intervals. k denotes the number of such intervals, and a_j and a_{j+1}, for $j=1, 2, \cdots, k$ are the beginning and the end of the jth interval $[a_j, a_{j+1}]$, respectively. In the example in Fig. 3, there are six such intervals; the time instants $a_1, a_2, \cdots, a_7$ are 0, 3, 7, 9, 13, 15, and 16, respectively.

	r_i	d_i	τ_i	m_i	o_i
T_1	0	7	5	3	2
T_2	3	12	7	4	3
T_3	2	14	6	2	4
T_4	5	16	6	4	2
T_5	5	18	3	2	1
T_6	10	19	4	1	3

(a)



(b)

FIG. 3. *An example illustrating Algorithm* F.

(2) The interval $[a_{k+1}, a_{k+2}]$. Let a_{k+2} be the latest finishing time of all tasks in the schedule S_t. Clearly, $a_{k+2} \geqq a_{k+1}$. In our example, a_{k+2} is 19.

(3) $M(j)$, for $j=1, 2, \cdots, k$. $M(j)$ is the mandatory task that is scheduled in $[a_j, a_{j+1}]$ in S_m. Let $T(j)$ be the corresponding task.

(4) X_j, for $j=1, 2, \cdots, k+1$. X_j is the set of tasks scheduled in $[a_j, a_{j+1}]$ according to the scheduled S_t.

(5) $L_m(j)$, $j=1, 2, \cdots, k$. $L_m(j)$ is the total amount of processor time assigned to the task $M(j)$ in the interval $[a_j, a_{j+1}]$ and all later intervals according to the schedule S_m. $L_m(j)$ is computed from S_m. In our example in Fig. 3, the values of $L_m(j)$ are 3, 4, 2, 4, 2, and 1 for j equals to $1, 2, \cdots$, and 6, respectively.

(6) $L_t(j)$, $j=1, 2, \cdots, k$. $L_t(j)$ is the total amount of processor time assigned to the task $T(j)$ in the interval $[a_j, a_{j+1}]$ and all later intervals according to the schedule S_t. $L_t(j)$ is initially given by S_t and is later modified during the adjustment process. In our example, the initial values of $L_t(j)$ are 5, 7, 2, 2, 2, and 1, for j equals to $1, 2, \cdots, 6$, respectively.

The operations of Step 3 are described by the pseudocode in Fig. 4. Step 3 modifies S_t as follows: The last segment of S_t in the interval $[a_{k+1}, a_{k+2}]$ is left unchanged. We examine in turn, for $j=k, k-1, \cdots, 1$, the value of $L_t(j)$, the total processor time assigned to the task $T(j)$ in the interval $[a_j, a_{j+1}]$ and all later intervals according to the schedule S_t. If $L_t(j)$ is equal to or larger than the total processor time $L_m(j)$ assigned to this task in these intervals according to S_m, the segment of S_t in $[a_j, a_{j+1}]$ is left unchanged. Otherwise, let $\Delta = L_m(j) - L_t(j)$. We assign Δ additional units of processor time in $[a_j, a_{j+1}]$ to $T(j)$. These units may be originally assigned to some other tasks in X_j. We decrease the amounts of processor time assigned to them in this interval and update the values of $L_t(i)$, for $i=1, 2, \cdots, j$, for all the tasks affected by this reassignment accordingly. This reassignment can always be done because Δ is less than or equal to $a_{j+1} - a_j$ and $T(j)$ is ready in the interval.

```
Step 3: processor time adjustment process
  L_m(j) and initial values of L_t(j) for j = 1, 2, ..., k are given by S_m and S_t, respectively.
  begin
      j = k
      while (1 ≦ j ≦ k)
          if (L_m(j) > L_t(j))
              Δ = L_m(j) − L_t(j)
              Assign Δ units of processor time in [a_j, a_{j+1}] to T(j);
              Reduce the amounts of processor time assigned to other tasks in
                  X_j in [a_j, a_{j+1}] by Δ units to accomplish this reassignment.
              Update the values of L_t(1), L_t(2), ..., L_t(j) and modify S_t.
          endif
          j = j − 1
      endwhile
      S_o = S_t
  end Step 3
```

FIG. 4. *Pseudocode of Step* 3 *of Algorithm* F.

In the example in Fig. 3, $L_t(6)$ and $L_t(5)$ are left unchanged in the processor time adjustment process because they are equal to $L_m(6)$ and $L_m(5)$, respectively. $L_t(4)$ is 2 while $L_m(4)$ is 4; therefore, two additional units of processor time is assigned to $T(j)$, which is T_4. These two units of time are taken from T_2. T_2 has three units of

processor time in the interval [9, 13] before the reassignment and only one unit after the reassignment. The new values of $L_t(j)$ are 5, 5, 2, and 4 for $j=1$, 2, 3, and 4, respectively. Similarly, we compare $L_t(3)$ and $L_m(3)$, and so on.

3.2. Optimality of Algorithm F. The complexity of Algorithm F is the same as that of the ED Algorithm, that is, $O(n \log n)$. To show that Algorithm F is optimal, when $w_i=1$ for all i, we need the following lemma. Let $\sigma(T(j), t)$ and $\sigma(T(j), m)$ be the total amounts of processor time assigned to the task $T(j)$ in the intervals prior to a_j in the schedules S_t and S_m, respectively.

LEMMA 1. *$\sigma(T(j), t) \leqq \sigma(T(j), m)$ for all $j=1, 2, \cdots, k$.*

Proof. Both S_t and S_m are obtained using the ED Algorithm. Suppose that $\sigma(T(j), t) > \sigma(T(j), m)$ for some j. $T(j)$ must be scheduled in some interval $[t, t']$ prior to a_j according to S_t but not according to S_m. $T(j)$ is ready in $[t, t']$; moreover, it is the task with the earliest deadline, and hence the highest priority, in the interval. Therefore, a portion of $T(j)$ that is scheduled in $[a_j, a_{j+1}]$ according to S_m should be scheduled earlier in $[t, t']$. This contradicts the fact that S_m is an earliest-deadline-first schedule. □

THEOREM 2. *Algorithm* F *is optimal when the tasks have identical weights.*

Proof. If Algorithm F fails to find a feasible schedule of the given task set **T**, that is, the schedule of the mandatory set **M** found in Step 2 is imprecise, no feasible schedule of **T** exists. This fact follows from the optimality of the earliest-deadline-first algorithm [10].

On the other hand if feasible schedules of **T** exist, Algorithm F will find an optimal one that minimizes the total error of **T**. That this statement is true follows from the following four facts, which we now show are true:

(1) In both S_m and S_t, every task is scheduled between its ready time and deadline. After the adjustment process, this remains to be true.

(2) The processor time assignment in S_t of any task $T(j)$ is adjusted when we find that $T(j)$ is assigned insufficient time in S_t for $M(j)$ to complete. After the interval $[a_j, a_{j+1}]$ is processed in Step 3, we guarantee that in the time intervals $[a_j, a_{j+1}]$, $[a_{j+1}, a_{j+2}], \cdots, [a_{k+1}, a_{k+2}]$, the total amount of processor time assigned to $T(j)$ in S_t is at least equal to that in S_m. Since $M(j)$ is precisely scheduled in S_m, there is sufficient processor time in the adjusted schedule for it to complete, that is, the resultant schedule is valid.

(3) Lemma 1 states that the amount of processor time assigned to $T(j)$ prior to a_j in the schedule S_t is less than or equal to that in the schedule S_m. Therefore, we will never assign too much processor time in Step 3 to any task $T(j)$ for the total amount to exceed its processing time.

(4) The schedule S_t minimizes the total error of **T**. Throughout Step 3, no additional idle time is introduced. The resultant schedule, therefore, minimizes the total error of **T**. □

4. Scheduling tasks with difference weights to minimize total error. In this section, we consider the case where tasks in **T** have different weights. We number the tasks in **T** according to their weights such that $w_1 \geqq w_2 \geqq \cdots \geqq w_n$. Let σ_i^o denote the processing time of the scheduled portion of the optional task O_i in some optimal schedules of **T**. In other words, σ_i^o is the total amount of processor time assigned to O_i in these schedules. The LWF (*largest-weight-first*) algorithm, described by the pseudocode in Fig. 5, first finds the values of σ_i^o for all i in a nonempty subset of optimal schedules and then finds an optimal schedule in this subset. Lemmas 2 and 3 provide the basis for the procedure used to find σ_i^o.

```
LWF Algorithm
  Tasks are indexed so that w_1 ≧ w_2 ≧ · · · ≧ w_n
  begin
    Use the ED Algorithm to find a schedule S_m of M.
    If S_m is not precise, stop; the task set T cannot be feasibly scheduled.
    else
      The mandatory set M'(={M'_1, M'_2, · · · , M'_n}) = M
      i = 1
      while (1 ≦ i ≦ n)
        Use Algorithm F to find an optimal schedule S^i_o of M' ∪ {O_i};
          O'_i = the portion of O_i scheduled in S^i_o
        M'_i = M_i ∪ O'_i
        i = i + 1
      endwhile
      The optimal schedule sought is S^n_o
    endif
  end Algorithm LWF
```

FIG. 5. *Pseudocode of the* LWF *Algorithm.*

Let $\mathbf{T}_i = \{M_1, M_2, \cdots, M_{i-1}, T_i, T_{i+1}, \cdots T_n\}$ be the task set in which the first $i-1$ tasks have no optional tasks. Let S^i_o be an optimal schedule of the set $\mathbf{M} \cup \{O_i\}$ found using the Algorithm F. In S^i_o all tasks in $\mathbf{M}$ are precisely scheduled, and since there is only one optional task, the total (weighted) error is minimized. In other words, the amount of processor time σ^o_i assigned to O_i in S^i_o is as large as feasible.

LEMMA 2. *The amount of processor time σ_i assigned to the task O_i in any optimal schedule of the task set $\mathbf{T}_i$ is equal to or less than σ^o_i.*

The proof of this lemma, being straightforward, is omitted. Let $\mathbf{S}_i$ denote the set of optimal schedules of $\mathbf{T}_i$ in each of which the amount of processor time assigned to O_i is σ^o_i.

LEMMA 3. *The set $\mathbf{S}_i$ of optimal schedules of $\mathbf{T}_i$ is nonempty if the amount of processor time assigned to O_i in the schedule S^i_o is σ^o_i.*

Proof. Suppose that in S^i_o the total amount of processor time assigned to O_i is equal to σ^o_i; however, $\mathbf{S}_i$ is empty. Let S_t be an optimal schedule of $\mathbf{T}_i$ in which the total amount of processor time σ_i assigned to the optimal task O_i is not equal to σ^o_i. Without loss of generality, let S_t be an earliest-deadline-first schedule. We now show that it is possible to transform the schedule S_t into another optimal schedule S_o in which the amount of processor time assigned to O_i is σ^o_i and, thus, lead to a contradiction to the supposition that $\mathbf{S}_i$ is empty.

Let S_m be an earliest-deadline-first schedule obtained by scheduling, according to the ED Algorithm, the mandatory set $\mathbf{M}$ and the portion of O_i that is scheduled in S^i_o. The transformation of S_t into S_o is done using S_m as a template. The processor time adjustment procedure used for this transformation is essentially the same as the one used in Step 3 of Algorithm F; this step is described in Fig. 4. The only difference is that we now treat M_i as well as the portion of O_i scheduled in S_m as a mandatory task M'_i.

The proof that the processor time adjustment procedure in Fig. 4 leads to an optimal schedule S_o of $\mathbf{T}_i$ in which the amount of processor time assigned to O_i is equal to σ^o_i is essentially the same as the proof of Theorem 2. Specifically, (1)-(3) in the proof of Theorem 2 remain to be true. To show that (4) the total error is not

increased by the processor time adjustment procedure, we note that no additional idle time is introduced by the procedure. Before and after this adjustment is made, the amounts of processor time assigned to O_i are σ_i and σ_i^o, respectively. Because of Lemma 2, $\sigma_i^o \geqq \sigma_i$. This additional $\sigma_i^o - \sigma_i$ units of processor time assigned to O_i in S_o is obtained by reducing the amounts of processor time assigned to the optimal tasks $O_{i+1}, O_{i+2}, \cdots, O_n$. Since T_i has the largest weight among the tasks $T_i, T_{i+1}, \cdots, T_n$, the total error of the schedule S_o is at most as large as that of the schedule S_t. However, S_o is in the set $\mathbf{S}_i$, which contradicts the supposition that $\mathbf{S}_i$ is empty. □

Lemma 3 allows us to determine the amounts of processor time that should be assigned to the optional tasks in **T** in an optimal schedule with the minimum total error. After using the ED Algorithm to verify that the task set **T** can be feasibly scheduled, we determine these amounts as follows. First, we use the Algorithm F to schedule the set of tasks $\mathbf{M} \cup \{O_1\}$. Again, T_1 is the task with the largest weight; in the resultant schedule S_o^1, its optional task is assigned σ_1^o units of processor time. From Lemma 3, we know that there are optimal schedules of **T** in which O_1 is assigned σ_1^o units of processor time. We commit ourselves to find one of these schedules by combining M_1 and the portion O_1' of O_1 that is scheduled in S_o^1 into a task M_1'. The task M_1' is treated as a mandatory task in the subsequent steps. In the next step, we again use Algorithm F to schedule the task set $\{M_1', M_2, \cdots, M_n\} \cup \{O_2\}$. Let O_2' be the portion, with processing time σ_2^o, of the optional task O_2 that is scheduled in the resultant optimal schedule S_o^2. Again, Lemma 3 states that there are optimal schedules of **T** in which the amounts of processor time assigned to O_1 and O_2 are σ_1^o and σ_2^o, respectively. We commit ourselves to find one of these schedules by combining M_2 and O_2' into the mandatory task M_2'. We repeat these steps for $i = 3, 4, \cdots, n$ until all σ_i^o are found. The schedule S_o^n found in the last step is an optimal schedule of **T** with minimum total error. Again, this algorithm is called the LWF *Algorithm* and is described in Fig. 5. Its time complexity is $O(n^2)$.

THEOREM 3. *The* LWF *Algorithm is optimal.*

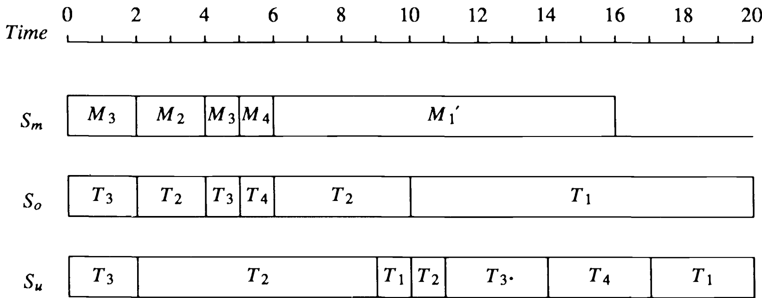
Proof. The proof by induction of this theorem follows immediately from Lemma 3 and the definition of the LWF Algorithm. □

Figure 6 shows an example. There are four tasks, and their weights are listed in Fig. 6(a). The schedule S_m of $\mathbf{M} \cup \{O_1\}$ produced by Algorithm F is shown in Fig. 6(b). We commit ourselves to finding an optimal schedule in which the amount of processor time assigned to O_1 is six. This schedule is an earliest-deadline-first schedule. It is used in the second step as a template to find an optimal schedule of the task set $\{M_1', T_2, M_3, M_4\}$. This resultant schedule S_o is shown in Fig. 6(b). The total error of the tasks is 25. Also shown is a schedule S_u that minimizes the unweighted total error; in this schedule the amount of processor time assigned to T_1 is only the minimum four units required to complete M_1.

5. Scheduling with 0/1 constraints. A schedule is said to satisfy the 0/1 constraint if according to the schedule every optional task is either completed in the traditional sense or discarded entirely. Hereafter, the expression that an optional task O_i is scheduled means that it is assigned o_i units of processor time. We consider only the case where all the tasks have identical weights. In this section, we first show that the problem of scheduling to meet the 0/1 constraint and timing constraints as well as to minimize the total error is NP-complete when the optional tasks have arbitrary processing times. We then present two algorithms for finding optimal schedules in the special case where all optional tasks have equal processing time. An optimal schedule is a feasible schedule in which the number of discarded optional tasks is minimum.

	r_i	d_i	τ_i	m_i	o_i	w_i
T_1	6	20	10	4	6	4
T_2	2	12	9	2	7	3
T_3	0	14	8	3	5	2
T_4	5	17	13	1	12	1

(a)



(b)

FIG. 6. *An example illustrating the* LWF Algorithm.

THEOREM 4. *The problem of scheduling tasks whose optional tasks have arbitrary processing times to meet* 0/1 *constraints and timing constraints and to minimize total error is* NP-*complete.*

Proof. It suffices for us to show that the problem is NP-complete in the following special case. In the set $\mathbf{T}=\{T_1, T_2, \cdots, T_n\}$, the processing times of all mandatory tasks are equal to zero. Let $L=\sum_{i=1}^{n} o_i$ be the sum of the processing times of all the optional tasks in $\mathbf{T}$. The ready times of all tasks in $\mathbf{T}$ are equal to zero, and their deadlines are $L/2$. That the problem is in NP is obvious.

To show that the problem is NP-complete, we transform the partition problem, which is known to be NP-complete, into our problem. In the partition problem, we have a set B of n elements; each element $b_i \in B$ has a size $s(b_i) \in Z^+$. Corresponding to each element b_i there is an optional task in our task set; the processing time of this optional task is $s(b_i)$. If we can find a subset $B' \subseteq B$ such that

$$\sum_{b_i \in B'} s(b_i) = \sum_{b_i \in B-B'} s(b_i),$$

then we can find a schedule of $\mathbf{T}$ such that the processor never idles and the total error is $L/2$, the minimum possible value. Optional tasks corresponding to the elements in this subset are discarded. On the other hand, if we cannot find a subset B' satisfying the above condition, we cannot find a schedule with minimum total error $L/2$. □

Hereafter, in this section, we confine our attention to the case where $O_i=\delta$, for $i=1, 2, \cdots, n$; δ is an arbitrary rational number. Our algorithm for scheduling with 0/1 constraint uses the following strategy: Each task and, hence, each optional task in $\mathbf{T}$ is assigned a *preference*, an integer value determined from the ready times and deadlines of the tasks in $\mathbf{T}$. We first determine whether to schedule the optional task with the highest (that is, the largest) preference, then determine whether to schedule the optional task with the second highest preference, and so on. In particular, let p_i denote the preference given to the task T_i. For any two tasks T_i and T_j, we have

(1) $p_i > p_j$ if $r_i \leqq r_j$ and $d_i > d_j$, or $r_i < r_j$ and $d_i \geqq d_j$, that is, the interval $[r_i, d_i]$ contains the interval $[r_j, d_j]$;

(2) $p_i > p_j$ if $r_i < r_j$ and $d_i < d_j$; and

(3) $p_i = p_j$ if $r_i = r_j$ and $d_i = d_j$.

We will return to justify this choice of giving preferences to different tasks. Without loss of generality, we index the tasks in **T** such that $p_1 \geqq p_2 \geqq \cdots \geqq p_n$.

5.1. The LDF Algorithm for scheduling tasks with the same ready time. When all tasks have the same ready time, the tasks with later deadlines have higher preferences. In this case, the LDF (*latest-deadline-first*) Algorithm shown in Fig. 7 can be used to find an optimal schedule with the minimum number of discarded optional tasks. According to the LDF Algorithm, a feasible, precise schedule S_m of the mandatory tasks is first constructed using the ED Algorithm. Since the ready times of all the tasks are the same, the mandatory tasks are never preempted in S_m. Let $t_1, t_2, \cdots, t_n$ be the time instants at which $M_1, M_2, \cdots, M_n$ completes according to S_m. In the next step, we try to schedule O_1, then O_2, and so on by readjusting the processor time assignments of the tasks. This step is described by the pseudocode in Fig. 7. The complexity of this algorithm is $O(n \log n)$.

```
LDF Algorithm
  Step 1: Use the ED Algorithm to find a schedule S_m of the mandatory set M. If S_m is not precise, the
      task set T cannot be feasibly scheduled; stop. Otherwise, carry out Step 2.
  Step 2: t_1, t_2, ..., t_n are the completion times of M_1, M_2, ..., M_n, respectively.
    begin
        d_{n+1} = ready time of all tasks
        j = 1
        real_deadline = d_1
        while (1 ≦ j ≦ n)
            if(real_deadline − t_j) ≧ δ
                Schedule O_j and assign [real_deadline − m_j − δ, real_deadline] to T_j
                real_deadline = min (real_deadline − m_j − δ, d_{j+1})
            else
                Discard O_j; assign [real_deadline − m_j, real_deadline] to T_j
                real_deadline = min (real_deadline − m_j, d_{j+1})
            endif
            j = j + 1
        endwhile
    end Step 2
```

FIG. 7. *Pseudocode of the* LDF *Algorithm.*

THEOREM 5. *The* LDF *Algorithm is optimal when all tasks have the same ready time.*

Proof. Given an arbitrary optimal schedule in which O_i is scheduled but O_j is discarded, and $p_j > p_i$, that is, $d_j > d_i$, we can transform this schedule into an optimal schedule in which O_j is scheduled and O_i is discarded. This is done by reassigning the time interval assigned to O_i in the given schedule to O_j instead. This reassignment is possible because $d_j > d_i$. It follows that any optimal schedule can be transformed into a schedule constructed by the LDF Algorithm. □

5.2. The DFS Algorithm for scheduling tasks with arbitrary ready times. The DFS (*depth-first-search*) Algorithm for scheduling tasks with arbitrary ready times is described by the pseudocode in Fig. 8. Lemma 4 provides the basis for this algorithm.

```
DFS Algorithm
    Assign preference p_i to tasks; the tasks are indexed such that p_1 ≧ p_2 ≧ ··· ≧ p_n.
    i = 1
    schedulables = φ
    M' = M
    while (1 ≦ i ≦ n)
        Use the ED Algorithm to find a schedule S^i_m of the tasks in M' ∪ {O_i}
        If (S^i_m is a precise schedule)
            schedulables = schedulables ∪ {O_i}
            Make the entire task T_i mandatory; M' = M' ∪ {O_i}
        endif
        i = i + 1
    endwhile
    Use the ED Algorithm to find a precise schedule of M ∪ schedulables
end Algorithm DFS
```

FIG. 8. *Pseudocode of the* DFS *Algorithm.*

Again $\mathbf{T}_i = \{M_1, M_2, \cdots, M_{i-1}, T_i, T_{i+1}, \cdots, T_n\}$ is the task set in which the first $i-1$ tasks has no optional tasks.

LEMMA 4. *There is a feasible, precise schedule of the task set* $\mathbf{M} \cup \{O_i\}$ *if and only if there exists an optimal schedule* S_o^i *of* $\mathbf{T}_i$ *in which* O_i *is scheduled.*

Proof. Obviously, if there is an optimal schedule S_o^i, the set $\mathbf{M} \cup \{O_i\}$ has a feasible and precise schedule. Suppose that the set $\mathbf{M} \cup \{O_i\}$ has a feasible and precise schedule, but there is no optimal schedule S_o^i in which O_i is scheduled. Let j be the smallest integer such that there is an optimal schedule S_o^j in which $O_i, O_{i+1}, \cdots, O_{j-1}$ are discarded and O_j is scheduled. If no such S_o^j exists, the feasible, precise schedule of $\mathbf{M} \cup \{O_i\}$ is optimal, we have a contradiction to the supposition that S_o^i does not exist. Therefore, there exists an S_o^j.

Let x be any index such that $i < x < j$. Given that there is an S_o^j, we note that it is not possible for the ready time and deadline of T_j to be such that either (1) $r_j = r_x$ and $d_j = d_x$ or (2) (r_j, d_j) is contained in (r_x, d_x) for any task T_x. If either (1) or (2) is true, there is an optimal schedule in which O_x is scheduled. This contradicts that j is the smallest index for S_o^j to exist. Therefore the ready times and deadlines of T_x and T_j must be as shown in Fig. 9. In addition to T_x and T_j, the possible values of ready times and deadlines of the other tasks are also shown. The arrow indicates the direction of increasing preferences.

We need to consider two cases: either it is possible to adjust the processor time assignments of $T_i, \cdots, T_x, \cdots, T_j$ so that O_x is scheduled instead of O_j or it is impossible to do so. In the former case, we have a contradiction to the fact that j is the smallest integer such that an optimal schedule S_o^i exists. In the latter case, we also cannot adjust the processor time assignments of these tasks so that O_i is scheduled and O_j is discarded. It is not possible for a feasible, precise schedule of $\mathbf{M} \cup \{O_i\}$ to exist. We again have a contradiction. Therefore, we can conclude that S_o^i exists. □

Lemma 4 allows us to determine whether the subset of optimal schedules in which the optional task O_1 is scheduled is empty. If it is empty, we decide to discard O_1. If it is not empty, we decide to schedule O_1. After this decision is made, we then proceed to determine whether the subset of optimal schedules in which both O_1 and O_2 are scheduled is empty. If it is, we decide to discard O_2; otherwise, we decide to schedule O_2, and so on. The DFS Algorithm works in this manner to choose the subset of

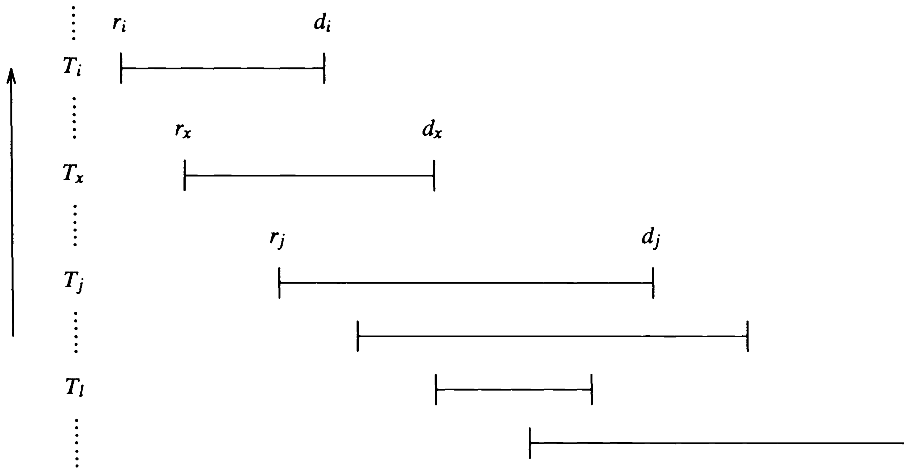


FIG. 9. *Ready times and deadlines of* T_i *and* T_j.

optional tasks to be scheduled and to construct a schedule of $\mathbf{T}=\{T_1, T_2, \cdots, T_n\}$ that satisfies the 0/1 constraint and minimizes the number of discarded optional tasks. Again, the algorithm is described by the pseudocode in Fig. 8. The complexity of this algorithm is $O(n^2)$.

THEOREM 6. *The* DFS *Algorithm is optimal.*

Proof. We prove this theorem by induction. Clearly, if $\mathbf{M}\cup\{O_1\}$ has a feasible and precise schedule, O_1 is schedulable according to Lemma 4. Otherwise O_1 should be discarded, since the subset of optimal schedules of $\mathbf{T}$ in which O_1 is scheduled is empty. Suppose that the theorem is true for some j. When the decision in regards with whether O_{j+1} should be scheduled or discarded is to be made, we have already decided about $O_1, O_2, \cdots, O_j$. In making the new decision, we try to find a feasible, precise schedule of $\{M'_1, M'_2, \cdots, M'_j, T_{j+1}, M_{j+2}, \cdots, M_n\}$ where the mandatory task M'_i is M_i if O_i is to be discarded and is T_i if O_i is to be scheduled, for $i\leqq j$. Again, we can use Lemma 4 to determine whether O_{j+1} should be scheduled or discarded. □

6. Summary. We present here a fast algorithm, called Algorithm F, for finding preemptive, feasible schedules of n dependent tasks with rational ready times, deadlines and processing times. Our criterion of optimality is that the algorithm guarantees to find a feasible schedule if such schedule exists, and, among all feasible schedules, the algorithm finds one with the minimum total error. Algorithm F is optimal when used to schedule dependent tasks with identical weights on uniprocessor systems. By applying McNaughton's rule [11], it can be modified to optimally schedule independent tasks with identical weights on an identical multiprocessor system containing v processors. The complexity of Algorithm F is $O(n \log n)$ in the case of uniprocessor systems and is $O(vn+n \log n)$ in the case of identical multiprocessor systems. For the case of different task weights, the LWF Algorithm with time complexity $O(n^2)$ solves this problem.

We also consider here the problem of scheduling tasks with 0/1 constraint, that is, every optional task is either executed to completion or discarded entirely. This general problem is shown to be NP-complete. In the special case when all optional tasks have equal processing time, the DFS Algorithm finds optimal schedules of dependent tasks on uniprocessor systems. Again, a schedule is said to be optimal if it satisfies the 0/1 constraint, is feasible, and minimizes the number of discarded optional

tasks. The complexity of the DFS Algorithm is $O(n^2)$. If the ready times of tasks are the same, the simpler LDF Algorithm with complexity $O(n \log n)$ can be used instead to find optimal schedules.

An extension of the problems considered here is that of finding approximate algorithms to solve the general problem of scheduling with 0/1 constraints when the optional tasks have arbitrary processing times. When the criterion of optimality is the number of discarded optional tasks, regardless of their processing times, a good strategy is to give higher preference to optional tasks with shorter processing times and try to schedule them first. Alternatively, a good strategy for scheduling tasks with 0/1 constraint to minimize total error is to give tasks with longer processing times higher preference. A paper on the worst-case performance bounds of these approximate algorithms is in preparation.

REFERENCES

[1] J. W. S. LIU, K. J. LIN, AND S. NATARAJAN, *Scheduling real-time, periodic jobs using imprecise results*, in Proc. 8th IEEE Real-Time Systems Symposium, San Jose, CA, December 1987.

[2] J. Y. CHUNG AND J. W. S. LIU, *Performance of algorithms for scheduling periodic jobs to minimize average error*, in Proc. 9th IEEE Real-Time Systems Symposium, Huntsville, AL, December 1988.

[3] J. Y. CHUNG, J. W. S. LIU, AND K. J. LIN, *Scheduling periodic jobs that allow imprecise results*, IEEE Trans. Comput., 39 (1990), pp. 1156–1173.

[4] K. J. LIN, S. NATARAJAN, J. W. S. LIU, AND T. KRAUSKOPF, *Concord: A system of imprecise computations*, in Proc. 1987 IEEE Compsac, Tokyo, Japan, October 1987.

[5] W. K. SHIH, J. W. S. LIU, J. Y. CHUNG, AND D. W. GILLIES, *Scheduling tasks with ready times and deadlines to minimize average error*, ACM Operating Systems Review, July 1989.

[6] J. BLAZEWICZ AND G. FINKE, *Minimizing mean weighted execution time loss on identical and uniform processors*, Inform. Process. Lett., 24 (1987), pp. 259–263.

[7] A. L. LIESTMAN AND R. H. CAMPBELL, *A fault-tolerant scheduling problem*, IEEE Trans. Software Engrg., 12 (1986), pp. 1089–1095.

[8] E. K. P. CHONG AND W. ZHAO, *Performance evaluation of scheduling algorithms for imprecise computer systems*, Tech. Report, Department of Computer Science, University of Adelaide, SA 5001, Adelaide, Australia, September 1988.

[9] ———, *User controlled optimization in task scheduling for imprecise computer systems*, Tech. Report, Department of Computer Science, University of Adelaide, SA 5001, Adelaide, Australia, October 1988.

[10] E. L. LAWLER AND J. M. MOORE, *A functional equation and its application to resource allocation and scheduling problem*, Management Sci., 16 (1969), pp. 77–84.

[11] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Sci., 12 (1959), pp. 1–12.

SIAM J. COMPUT.
Vol. 20, No. 3, pp. 553-590, June 1991
© 1991 Society for Industrial and Applied Mathematics
011

BOOLEAN FUNCTIONS, INVARIANCE GROUPS, AND PARALLEL COMPLEXITY*

PETER CLOTE† AND EVANGELOS KRANAKIS‡

Abstract. This paper studies the invariance groups $\mathbf{S}(f)$ of boolean functions $f \in \mathbf{B}_n$ (i.e., $f:\{0,1\}^n \rightarrow \{0,1\}$) on n variables, i.e., the set of all permutations on n elements which leave f invariant. After building intuition by presenting several examples that suggest relations between algebraic properties of groups and computational complexity of languages, necessary and sufficient conditions are given via Pólya's cycle index for an arbitrary finite permutation group to be of the form $\mathbf{S}(f)$, for some $f \in \mathbf{B}_n$. It is shown that asymptotically "almost all" boolean functions have trivial invariance groups. For cyclic groups $G \leqq \mathbf{S}_n$ a logspace algorithm for determining whether the given group is of the form $\mathbf{S}(f)$, for some $f \in \mathbf{B}_n$ is given. The applicability of group theoretic techniques in the study of the parallel complexity of languages is demonstrated. For any language L let L_n be the characteristic function of the set of all strings in L which have length exactly n and let $\mathbf{S}_n(L)$ be the invariance group of L_n. The index $|\mathbf{S}_n : \mathbf{S}_n(L)|$ are considered as a function of n and the class of languages whose index is polynomial in n is studied. Bochert's lower bound on the index of primitive permutation groups is used together with the O'Nan-Scott theorem, a deep result in the classification of finite simple groups, in order to show that any language with polynomial index is in (nonuniform) TC^0 and hence in (nonuniform) NC^1. As a corollary, an extension is given of a result of Fagin-Klawe-Pippenger-Stockmeyer, giving necessary and sufficient conditions for a language with polynomial index to be computable by a constant depth polynomial size circuit family. As another corollary, it is shown that the problem of "weight-swapping" for a sequence of groups of polynomial index is in (nonuniform) NC^1.

Key words. abelian group, boolean function, circuit, classification theory, cyclic-, dihedral-, hyperoctahedral-groups, index of a group, invariance group of boolean function, NC, parallel complexity, permutation group, Pólya cycle index, pumping lemma, representable group, regular language, symmetric boolean function, wreath product

AMS(MOS) subject classifications. 68Q15, 68Q25, 68Q45

1. Introduction. The aim of this paper is to study the invariance groups of boolean functions, provide efficient algorithms for determining the representability of a given group as the invariance group of a boolean function, and use group theoretic techniques in order to deduce results about the parallel complexity of formal languages.

Given n input values, each of which can assume one of two possible states 0, 1, a "module" M outputs a value which assumes one of the states 0, 1. The output of the module when the input values are $x_1, \cdots, x_n$ depends in general on the *order* of the inputs. There are certain permutations of the input states which leave the output state *invariant* or unchanged. For example, it may be that the output is independent of any permutation of the input states, in which case the given module is called symmetric. In general, for a given module, the set of permutations which, when applied to any set of input states, leave the output invariant is easily seen to form a permutation group.

* Received by the editors November 7, 1988; accepted for publication (in revised form) June 28, 1990. This article first appeared as an extended abstract in the Proceedings of the Fourth Annual IEEE Conference on Structure in Complexity Theory, 1989, pp. 55-65.

† Department of Computer Science, Boston College, Chestnut Hill, Massachusetts 02167. This author's research was supported in part by National Science Foundation grant DCR-8606165. Some of this research was performed while the author was visiting the Université de Paris VII, Equipe de Logique Mathématique, Centre Nationale de la Recherche Scientifique-UA 753, 2 Place Jussieu, Paris, France.

‡ Centrum voor Wiskunde en Informatica, P.O. Box 4079, 1009 AB Amsterdam, the Netherlands.

Formally, the operation performed by such an n-ary module M is usually represented by an n-ary boolean function[1] $f: 2^n \to 2$. For fixed n, let the set of all such n-ary boolean functions be denoted by $\mathbf{B}_n$. If the input states of the module are assigned the boolean values $x_1, \cdots, x_n$ then by definition $f(x_1, \cdots, x_n)$ is the value of the output state of the module M on input $x_1, \cdots, x_n$. Given such an n-ary boolean function f let $\mathbf{S}(f)$ be the set of all permutations on the n elements $1, 2, \cdots, n$ such that for all input values $(x_1, \cdots, x_n) \in 2^n$, $f(x_1, \cdots, x_n) = f(x_{\sigma(1)}, \cdots, x_{\sigma(n)})$. Clearly, the group $\mathbf{S}(f)$ equals the full symmetric group $\mathbf{S}_n$ exactly in the special case when the boolean function f is symmetric.

By a counting argument Lupanov, Shannon, and Strassen have shown that almost all boolean functions have exponential size circuit complexity. Despite this result, very little is known concerning specific languages or families of boolean functions. Our interest in the present study arose from attempting to use group theoretic techniques in order to generalize the simple observation that any family $\{f_n : f_n \in \mathbf{B}_n,\ n \in \mathbf{N}\}$ of symmetric boolean functions is computable by a logarithmic depth, polynomial size circuit family. Probabilistic techniques have been successfully used by several authors (Furst, Saxe, and Sipser [FSS84], Yao [Yao85], etc.) in order to obtain lower bounds on the size and/or depth of circuit families which compute certain symmetric languages (families of symmetric boolean functions). However, there are few results giving tight upper bounds, apart from the above cited fact that any family of symmetric boolean functions is computable by a nonuniform circuit family of logarithmic depth and polynomial size (formula size bounds have been obtained by various authors in this case). In this paper we indicate the applicability of group theory in obtaining upper bounds for the parallel complexity of families of boolean functions. Our work is different from, but somewhat related to, studies on the automorphism groups of error-correcting codes (e.g., kth order Reed-Muller codes, which are specific k-dimensional subspaces of 2^n [MS78]), as well as to work in [Har64] where group theoretic methods are used to calculate the number of nonequivalent boolean functions, where the equivalence relation is defined by $f \equiv g$ if and only if there exists $\sigma \in \mathbf{S}_n$ such that for all $x_1, \cdots, x_n \in \{0, 1\}$ $(f(x_1, \cdots, x_n) = g(x_{\sigma(1)}, \cdots, x_{\sigma(n)}))$.

In [FKL88] it was indicated how the classification theorem for finite simple groups could be applied to VLSI technology by giving an algorithm to minimize pin-count in a sequence of circuits. Here we consider the problem of placement of modules on a chip where permutation of input wires is allowed. It is expected that study of the invariance groups of boolean functions may lead to algorithms for optimizing space in VLSI design, e.g., knowledge that certain modules leading into a block can be permuted without changing the function computed.

It is interesting to point out that invariance groups are also relevant to the computability problem for boolean functions in anonymous networks as used in distributed computing. For example, we are interested in computing n-ary boolean functions in an n-node anonymous network $\mathcal{N}$. To compute the value of a given function f at the input $(b_1, \cdots, b_n)$ the processors $p_1, \cdots, p_n$ are initialized with the inputs $b_1, \cdots, b_n$, respectively. By exchanging messages through the links all the processors must eventually compute the same bit $b = f(b_1, \cdots, b_n)$. It has been the focus of several papers to determine and study networks for which

$$f \text{ is computable in } \mathcal{N} \Leftrightarrow S(f) \supseteq Aut(\mathcal{N}),$$

[1] Throughout the paper we identify a positive integer n with the set $\{0, 1, \cdots, n-1\}$, e.g., $2 = \{0, 1\}$; in general, however, we will prefer the set-notation when we want to emphasize the elements of the language under consideration.

where $Aut(\mathcal{N})$ denotes the group of automorphisms of $\mathcal{N}$. In fact, this is the case for several types of networks, like directed and unlabeled rings [ASW85], labeled tori [BB89], and labeled hypercubes [KK89].

1.1. Results of the paper. Following is an outline of the main results and contents of the paper. We begin in § 2 by providing some preliminary results regarding the size of the index of a permutation group. We remind the reader of the essential parts of Pólya's beautiful enumeration theory that will be used in the present study.

In §§ 3 and 4, to build intuition for the reader, we present a number of examples concerning the invariance groups of certain types of languages, such as palindromes, parentheses, and regular languages, and study the reverse problem of constructing languages realizing specific types of groups. We compute the invariance groups of Dyck palindrome languages and give an efficient algorithm for determining membership in the invariance group of regular languages. We show that each of the cyclic (for $n \neq 3, 4, 5$), dihedral, and hyperoctahedral sequences of groups are representable by regular languages and construct groups which cannot be represented by regular languages.

In § 5 we study the representation problem for general permutation groups. We define a subgroup $G \leqq \mathbf{S}_n$ to be *strongly representable* if G is the invariance group of an n-ary boolean function—i.e., there exists $f \in \mathbf{B}_n$ for which $G = \mathbf{S}(f)$. We distinguish between groups which are "strongly representable" and groups which are "isomorphic to strongly representable." In the latter case, we show that every permutation subgroup of $\mathbf{S}_n$ is isomorphic to a strongly representable group $\mathbf{S}(f)$, for some $f: 2^{n(\log n+1)} \rightarrow 2$; but as stated, this isomorphism is at the expense of increasing the number of variables in the boolean function from n to $n(\log n + 1)$. The problem is more interesting in the former case, where we give a necessary and sufficient condition in terms of the Pólya index, for an arbitrary subgroup of $\mathbf{S}_n$ to be of the form $\mathbf{S}(f)$, for some n-ary boolean function $f: 2^n \rightarrow 2$. Using the classification theorem for maximal permutation groups we show that "with few exceptions" (essentially, only the alternating group $\mathbf{A}_n$, for $n \geqq 10$) all maximal permutation groups on n letters are strongly representable. This contrasts with the fact that there are numerous nonrepresentable permutation groups. We also give a logspace algorithm which, on input of a cyclic group $G \leqq \mathbf{S}_n$, decides whether G is strongly representable, in which case it outputs a boolean function $f: 2^n \rightarrow 2$ such that $G = \mathbf{S}(f)$. Our last result in this section concerns asymptotics. For any sequence of nonidentity permutation groups $\langle G_n \leqq \mathbf{S}_n : n \geqq 1 \rangle$ we prove that

$$\lim_{n \to \infty} \frac{|\{f \in \mathbf{B}_n : \mathbf{S}(f) \geqq G_n\}|}{2^{2^n}} = 0.$$

It then immediately follows that asymptotically "almost all" boolean functions have a trivial invariance group; i.e., they are equal to the identity permutation group.

Given a language $L \subseteq \{0, 1\}^*$, let L_n be the characteristic function of the set of words of L of length exactly n. Section 6 is concerned with the complexity of languages of polynomial index, i.e., languages L for which there exists a polynomial $p(n)$ such that $|\mathbf{S}_n : \mathbf{S}_n(L)| \leqq p(n)$, where $\mathbf{S}_n(L)$ denotes the invariance group of the boolean function L_n. We study the closure properties of the class of these languages and apply the NC algorithm for permutation group membership of [BLS87] in order to show that languages of polynomial index are in (nonuniform) NC. By using the O'Nan–Scott theorem, a deep result in classification theory of finite simple groups, we improve the last result to show that any language of polynomial index is in (nonuniform) TC^0 and hence NC^1.

In [FKPS85], Fagin, Klawe, Pippenger, and Stockmeyer used group theoretic techniques together with the exponential size lower bound for constant depth circuits accepting parity [Yao85] to give a necessary and sufficient condition for a symmetric language $L \subseteq \{0,1\}^*$ to belong to AC^0; i.e., for L to be computable by a nonuniform circuit family of constant depth and polynomial size. Our characterization of languages of polynomial index allows an immediate extension of this result. Namely, for $L \subseteq \{0,1\}^*$ of polynomial index, L is in AC^0 if and only if the least number of input bits which must be set to a constant in order for the resulting language $L_n = L \cap \{0,1\}^n$ to be constant is polylogarithmic in n.

As mentioned in the introduction, we believe that group theoretic considerations may possibly play a role in VLSI design. In particular, knowledge of the invariance group of "modules" might allow minimization of the surface area for automated circuit layout. Toward a mathematical formalization of this idea, we introduce some notation. For any sequence $\mathbf{G} = \{G_n: G_n \leqq \mathbf{S}_n, n \in \mathbf{N}\}$ of permutation groups the problem SWAP ($\mathbf{G}$) is given by the following.

Input. $n \in \mathbf{N}$, $a_1, \cdots, a_n$ positive rationals.

Output. A permutation $\sigma \in G_n$ such that for all $1 \leqq i < n$, $a_{\sigma(i)} + a_{\sigma(i+1)} \leqq 2$, if such a permutation exists, and the response "NO" otherwise.

The intuition behind the problem SWAP ($\mathbf{G}$) is that the output wires of modules $M_1, \cdots, M_n$ are the inputs to module M, and that the invariance group of M is G_n. The "width" of module M_i is the rational number a_i. Modules M_i and M_j can be placed next to each other if they do not "overlap"; i.e., exactly when $a_i + a_j \leqq 2$, where we imagine an average size of 1 per module. Thus, the output for SWAP ($\mathbf{G}$) indicates whether there exists a permutation of the input modules M_i which does not change the output of M and which allows a layout of $M_{\sigma(1)}, \cdots, M_{\sigma(n)}$ without overlap. A simple application of our work yields an NC^1 algorithm for the problem SWAP ($\mathbf{G}$), where $\mathbf{G} = \{G_n: G_n \leqq \mathbf{S}_n, n \in \mathbf{N}\}$ is of polynomial index.

Recall that the stipulation of the layout problem is to find an optimal layout given a number of modules together with their connections. A popular algorithm that attempts to solve the layout problem is due to Kernighan and Lin [KL82] and partitions the chip into an upper and a lower half, swapping modules on either side, trying to minimize a certain parameter, then recursively partitioning simultaneously the top and bottom into left and right parts, swapping modules between left and right parts to minimize a parameter, etc. Our problem stipulation in SWAP is quite different: instead of being given a list of modules and their connections (including which input port of a target module), we allow the input ports of the target module to be swapped, provided that the resultant function is not changed.

Finally, in § 7, we discuss some open problems and give directions for further research.

An acquaintance with the standard results on group theory and finite permutation groups, as presented for example in [Hal57] and [Wie64], will be essential for an adequate understanding of the results of the present paper.

2. Preliminaries. Here we give some introductory definitions and results regarding permutation groups and complexity of circuits that will be used in our subsequent investigations. The three topics we will discuss are:

- the size of the group index,
- the size of the cycle index and its computation via Pólya's formula, and
- complexity of boolean functions with respect to the size and/or depth of boolean circuits computing them.

2.1. Index of a permutation group. In the sequel it will be convenient to think of permutations on the set $\{1, 2, \cdots, n\}$ as bijective mappings on the set of all positive integers such that $\sigma(k) = k$ for all $k > n$. Part of this paper is primarily concerned with "large" permutation subgroups of the full symmetric group. Let $\mathbf{S}_n$ denote the group of all permutations of n elements, and $\mathbf{A}_n$ be the subgroup of even permutations (also known as the alternating group on n letters). In general, for any nonempty set Ω let $\mathbf{S}_\Omega$ denote the set of all permutations of Ω. For any group G the symbol $H \leqq G$ means that H is a subgroup of G. Regarding the sizes of permutation groups the following theorem summarizes some known results on the sizes permutation groups.

THEOREM 1. *Let $H \leqq \mathbf{S}_n$ be a permutation group which does not contain $\mathbf{A}_n$.*

(1) $|\mathbf{S}_n : H| \geqq n$.

(2) *If the order of H is maximal then $|\mathbf{S}_n : H| = n$. In fact, for $n \neq 6$ the subgroups H of $\mathbf{S}_n$ with $|\mathbf{S}_n : H| = n$ are exactly the one point stabilizers of $\mathbf{S}_n$.*

(3) *If H is primitive then*

(*Bochert*) $|\mathbf{S}_n : H| \geqq [(n+1)/2]!$.

(*Praeger and Saxl*) $|H| < 4^n$.

(*Cameron*) *either H is a "known" group or* $|H| < n^{10 \log \log n}$.

Proof. For all three parts and further information, consult [Wie64], [Tzu82], as well as the references in [KL88] (in particular, the proof of (3) is very hard). Part (1) follows from the following claim.

CLAIM. *If H is a subgroup of G and $|G : H| = n$ then there exists a normal subgroup N of G such that $N \leqq H$ and $|G : N|$ divides $n!$.*

Indeed, consider the set $\Omega = \{Hg: g \in G\}$ of cosets of the quotient group G/H. By assumption, this set has size n. Let S_Ω be the group of permutations on Ω. For each $x \in G$ consider the permutation $\phi(x): \Omega \to \Omega$, where $\phi(x)(Hg) = Hgx$. Clearly, $\phi: G \to S_\Omega$ is a group homomorphism. Moreover, it is easy to see that

$$N := Ker(\phi) = \bigcap_{g \in G} H^g$$

is a normal subgroup of G, where $H^g = g^{-1}Hg$. By the homomorphism theorem, the order of the quotient group G/N divides the order of the permutation group S_Ω. This proves the claim.

Now let us prove (1) by the above claim there exists a normal subgroup N of $\mathbf{S}_n$ such that $N \leqq H$ and $|\mathbf{S}_n : N|$ divides $(n-1)!$. It follows that $N \neq 1$. Since the only normal subgroups of $\mathbf{S}_n$ are $\mathbf{A}_n$, $\mathbf{S}_n$, and 1, the result is clear. □

2.2. Cycle index of a permutation group. Let G be a permutation group on n elements. Define an equivalence relation $i \equiv j$ if and only if for some $\sigma \in G$, $\sigma(i) = j$. The equivalence classes under this equivalence relation are called orbits. Let $G_i = \{\sigma \in G: \sigma(i) = i\}$ be the stabilizer of i, and let i^G be the orbit of i. An elementary theorem asserts that $|G : G_i| = |i^G|$. Using this, we can obtain the well-known theorem of Burnside and Frobenius, which states that for any permutation group G on n elements, the number of orbits of G is equal to the average number of fixed points of a permutation $\sigma \in G$,

$$\omega_n(G) = \frac{1}{|G|} \sum_{\sigma \in G} |\{i: \sigma(i) = i\}|, \tag{1}$$

where $\omega_n(G)$ is the number of orbits of G [Com70]. Any permutation $\sigma \in \mathbf{S}_n$ can be identified with a permutation on 2^n defined as follows:

$$x = (x_1, \cdots, x_n) \to x^\sigma = (x_{\sigma(1)}, \cdots, x_{\sigma(n)}).$$

Hence, any permutation group G on n elements can also be thought of as a permutation group on the set 2^n. It follows from (1) that

$$|\{x^G: x \in 2^n\}| = \frac{1}{|G|} \sum_{\sigma \in G} |\{x \in 2^n: x^\sigma = x\}|,$$

where $x^G = \{x^\sigma: \sigma \in G\}$ is the orbit of x. We would like to find a more explicit formula for the right-hand side of the above equation. To do this, note that $x^\sigma = x$ if and only if x is invariant on the orbits of σ. It follows that $|\{x \in 2^n: x^\sigma = x\}| = 2^{o(\sigma)}$, where $o(\sigma)$ is the number of orbits of (the group generated by) σ. Using the fact that $o(\sigma) = c_1(\sigma) + \cdots + c_n(\sigma)$, where $c_i(\sigma)$ is the number of i-cycles in σ (i.e., in the cycle decomposition of σ), we obtain Pólya's formula:

$$(2) \qquad |\{x^G: x \in 2^n\}| = \frac{1}{|G|} \sum_{\sigma \in G} 2^{o(\sigma)} = \frac{1}{|G|} \sum_{\sigma \in G} 2^{c_1(\sigma) + \cdots + c_n(\sigma)}.$$

The number $|\{x^G: x \in 2^n\}|$ is called the **cycle index** of the permutation group G and will be denoted by $\Theta(G)$. If we want to stress that G is a permutation group on n letters, then we write $\Theta_n(G)$, instead of $\Theta(G)$. For more information on Pólya's enumeration theory the reader should consult [Ber71] and [PR87].

Since the invariance group $\mathbf{S}(f)$ of a function $f \in \mathbf{B}_n$ contains G if and only if it is invariant on each of the different orbits x^G, $x \in 2^n$, we obtain that

$$|\{f \in \mathbf{B}_n: \mathbf{S}(f) \geqq G\}| = 2^{\Theta(G)}.$$

It is also not difficult to compare the size of $\Theta(G)$ and $|\mathbf{S}_n : G|$. Indeed, let $H \leqq G \leqq \mathbf{S}_n$. If

$$Hg_1, Hg_2, \cdots, Hg_k$$

are the distinct right cosets of G modulo H then for any $x \in 2^n$ we have that

$$x^G = x^{Hg_1} \cup x^{Hg_2} \cup \cdots \cup x^{Hg_k}.$$

It follows that $\Theta_n(H) \leqq \Theta_n(G) \cdot |G:H|$. Using the fact that $\Theta_n(\mathbf{S}_n) = n+1$ we obtain as a special case that $\Theta_n(G) \leqq (n+1)|\mathbf{S}_n : G|$. In addition, using a simple argument concerning the size of the orbits of a permutation group we obtain that if $\Delta_1, \cdots, \Delta_\omega$ are different orbits of the group $G \leqq \mathbf{S}_n$ acting on $\{1, 2, \cdots, n\}$ then

$$(|\Delta_1| + 1) \cdots (|\Delta_\omega| + 1) \leqq \Theta_n(G).$$

We summarize these results in the following useful theorem.

THEOREM 2. *For any permutation groups $H \leqq G \leqq \mathbf{S}_n$ we have*

(1) $\Theta_n(G) \leqq \Theta_n(H) \leqq \Theta_n(G) \cdot |G:H|$.

(2) $\Theta_n(G) \leqq (n+1) \cdot |\mathbf{S}_n : G|$.

(3) $n+1 \leqq \Theta_n(G) \leqq 2^n$.

(4) *If $\Delta_1, \cdots, \Delta_\omega$ are different orbits of G then* $(|\Delta_1|+1) \cdots (|\Delta_\omega|+1) \leqq \Theta_n(G)$.

It is easy to see that in general $|\mathbf{S}_n : G|$ and $\Theta_n(G)$ can diverge widely. For example, let $f(n) = n - \log n$ and let G be the group $\{\sigma \in \mathbf{S}_n: \forall i > f(n)(\sigma(i) = i)\}$. It is then clear that $\Theta_n(G) = (f(n)+1) \cdot 2^{\log n}$ is of order n^2, while $|\mathbf{S}_n : G|$ is of order $n^{\log n}$. Another simpler example is obtained when G is the identity subgroup of $\mathbf{S}_n$.

2.3. Circuits. An n-circuit α_n is a labeled, directed acyclic graph whose nodes are labeled by $x_1, \cdots, x_n$ (input bits), $\neg$, $\wedge$, $\vee$. The input nodes are of in-degree 0 and there is a unique output node whose out-degree is 0. The size $c(\alpha)$ of α_n is the number of internal (i.e., noninput) nodes, while the depth $d(\alpha)$ of α_n is the maximal length

of a path from an input node to the output node. A word $x \in \{0, 1\}^n$ is *accepted* by an n-circuit α_n if each input node labeled by x_i has as value the ith bit of x. An n-circuit α_n *recognizes* or *computes* a language $L_n \subseteq \{0, 1\}^n$ (respectively, boolean function $f \in \mathbf{B}_n$) if and only if for all words x in $\{0, 1\}^n$,

$$x \in L_n \text{ (respectively } f(x) = 1) \Leftrightarrow \alpha_n \text{ accepts } x.$$

A circuit family $\langle \alpha_n: \alpha_n$ is an n-circuit, $n \in \mathbf{N}\rangle$ *recognizes* or *computes* a language $L \subseteq \{0, 1\}^*$ if and only if for all n (α_n accepts $L \cap \{0, 1\}^n$). In this paper, we usually consider *nonuniform* circuit families as defined above—of course, such families can recognize nonrecursive languages. A circuit family $\langle \alpha_n: n \in \mathbf{N}\rangle$ is *logspace uniform* if there is a logspace computable function $F: 1^n \mapsto \bar{\alpha}_n$ for constructing the circuits. There are stronger and weaker uniformity notions. See [Coo85] for further discussion and for a survey of parallel complexity theory. The class *SIZE-DEPTH*(f, g) is the collection of languages accepted by a family $\langle \alpha_n: n \in \mathbf{N}\rangle$ where $c(\alpha_n) \leqq f(n)$ and $d(\alpha_n) \leqq g(n)$. The class AC^k (respectively, NC^k) is the collection of languages[2] belonging to *SIZE-DEPTH*$(n^{O(1)}, O(\log^k(n)))$ where the in-degree of nodes labeled by $\wedge$, $\vee$ is arbitrary (respectively, 2). Of importance to this paper is the class AC^0 of languages accepted by (nonuniform) circuit families of constant depth and polynomial size with *arbitrary fanin*, and the class NC^1 of languages accepted by (nonuniform) circuit families of logarithmic depth (and a fortiori polynomial size) with *fanin* 2. By unwinding a circuit into an equivalent boolean formula (circuit with fanout 1), NC^1 is easily seen to be the class of languages computable by (nonuniform) polynomial size boolean formulas. The class TC^0 is the collection of languages computable by (nonuniform) circuit families with constant depth and polynomial size, whose gates are arbitrary fanin *threshold* gates. NC is defined to be $\bigcup_{n \in \mathbf{N}} \mathrm{NC}^k$. Trivially, $\mathrm{NC}^k \subseteq \mathrm{AC}^k$, and by replacing an arbitrary fanin gate by a binary tree of fanin 2 gates, it is clear that $\mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}$. A language $L \subseteq \{0, 1\}^*$ is said to have (or be computable by) polynomial size circuits, denoted $L \in SIZE(n^{O(1)})$, if there is a circuit family $\langle \alpha_n: n \in \mathbf{N}\rangle$ where α_n computes the characteristic function of $L_n = L \cap \{0, 1\}^n$ and $c(\alpha_n) \leqq p(n)$ for some polynomial p. Note that $SIZE(n^{O(1)})$ is the same class, whether one considers arbitrary fanin or fanin 2 circuits. Since the out-degree of a node is arbitrary, partial computations may be reused; thus the circuit provides a model for parallel computation. Stockmeyer and Vishkin [SV84] have shown that AC^k is the class of languages computed in $O(\log^k(n))$ time with a polynomial number of processors on a *parallel random access machine* (PRAM).

For a boolean function $f: 2^n \to 2$, we define

$$c(f) = \min\{c(\alpha): \alpha \text{ computes } f\}$$

where α has fanin 2. The following results are well known (e.g., see [Sav76] or [Yab83]). In particular, we shall use the second fact in a later proof.

(1) For any symmetric function $f \in \mathbf{B}_n$, $c(f) = O(n)$.

(2) (Lupanov–Shannon–Strassen) $|\{f \in \mathbf{B}_n: c(f) < q\}| = O(q^{q+1})$.

(3) For any $\varepsilon > 0$, the ratio of $f \in \mathbf{B}_n$ such that $c(f) > (1-\varepsilon)2^{n-1}/n$ tends to 1 as $n \to \infty$.

3. Invariance groups of certain languages. The main objects of study in this paper are boolean functions and their invariance groups. Let $\mathbf{B}_{n,k}$ be the set of all k-valued

---

[2] Usually these classes are defined to be classes of functions rather than languages. Since we will not discuss function computations in this paper, we adopt the above definition.

functions $f: 2^n \to k$ on n boolean variables. If $k=2$ then we abbreviate $\mathbf{B}_{n,2}$ by $\mathbf{B}_n$. If $\mathbf{Z}_2$ denotes the finite two-element field then it is clear that

$$\mathbf{B}_n = \frac{\mathbf{Z}_2[x_1, \cdots, x_n]}{(x_i^2 - x_i,\ i = 1, 2, \cdots, n)}.$$

For $x = (x_1, \cdots, x_n) \in 2^n$ and $\sigma \in \mathbf{S}_n$, let $x^\sigma = (x_{\sigma(1)}, \cdots, x_{\sigma(n)})$. For any n-ary boolean function $f \in \mathbf{B}_n$ let f^σ be defined by

$$f^\sigma(x_1, \cdots, x_n) = f(x_{\sigma(1)}, \cdots, x_{\sigma(n)}).$$

The *invariance group* of f is defined by

$$\begin{aligned} \mathbf{S}(f) &= \{\sigma \in \mathbf{S}_n : f = f^\sigma\} \\ &= \{\sigma \in \mathbf{S}_n : \forall x \in 2^n f^\sigma(x_1, \cdots, x_n) = f(x_{\sigma(1)}, \cdots, x_{\sigma(n)})\}. \end{aligned}$$

If $K \subseteq \{0,1\}^n$ is a set of words of length n, then by abuse of notation we shall write $\mathbf{S}(K)$ for the invariance group of the characteristic function of the set K. If $L \subseteq \{0,1\}^*$ is a set of finite words and $n \geqq 1$ then $\mathbf{S}_n(L)$ denotes the invariance group of the n-ary boolean function L_n. Clearly, $\mathbf{S}(f)$, being nonempty and closed under multiplication, is a subgroup of $\mathbf{S}_n$.

Here we compute the invariance groups of well-known formal languages. We begin with the Dyck (or parenthesis) and palindrome languages and conclude with an "efficient" algorithm for computing the invariance group of regular languages.

3.1. Dyck languages. The semiDyck language D [Harr78] is defined as the least set of strings in the alphabet 0, 1 such that $\Lambda \in D$ and (for all $x, y \in D$) ($xy \in D$ and $0x1 \in D$). The semiDyck language is not regular, as can be seen from the fact that the elements 0^n give rise to infinitely many distinct equivalence classes in the right congruence relation for D. The Dyck languages D^r, $r \geqq 1$, are defined in the alphabet $\Sigma_r = \{0_i, 1_i : i = 1, \cdots, r\}$ in a similar fashion: D^r is the least set of strings in the alphabet Σ_r such that $\Lambda \in D^r$ and (for all $x, y \in D^r$) (for all $i \leqq r$) ($xy \in D^r \wedge 0_i x 1_i \in D^r$). Clearly, $D = D^1$. Next we determine the invariance group of the Dyck languages.

THEOREM 3. *For the Dyck language D^r defined above we have that*

$$\mathbf{S}_n(D^r) = \begin{cases} 1 & \text{if } n \text{ is odd or } r \geqq 2 \\ \langle (i, i+1) : i < n \text{ is even} \rangle & \text{if } n \text{ is even and } r = 1. \end{cases}$$

Proof. First, notice that D is a homomorphic image of D^r. The homomorphism $h_r : \Sigma_r \to \Sigma$ is defined by setting $h_r(b_i) = b$, where $b \in \{0,1\}$. It follows that for all strings x of length n, and all permutations $\sigma \in \mathbf{S}_n$, $h_r(x^\sigma) = (h_r(x))^\sigma$, which in turn implies that $\mathbf{S}_n(D^r) \subseteq \mathbf{S}_n(D)$. Now, if n is odd, then trivially $\mathbf{S}(D) = 1$ and so $\mathbf{S}(D^r) = 1$. Suppose that $n = 4$, $r = 2$, and, respectively, write "(", "[", ")", "]" in place of 0_1, 0_2, 1_1, 1_2. Then ([ ]) $\in D_4^2$, but (] [) $\notin D_4^2$. Similar examples can be constructed to verify that $\mathbf{S}(D^r) = 1$ for $2 \leqq r$. To prove the theorem, it is enough to show that, for n even,

$$\mathbf{S}_n(D) = \langle (i, i+1) : i < n \text{ is even} \rangle.$$

For any string $x = x_1 \cdots x_k$ let $l(x) = k$ be its length and $s(x)$ its signature, where

$$s(x) = \sum_{i=1}^{k} (-1)^{x_i}.$$

Then we can prove the following claims.

CLAIM 1. *For any string x, $x \in D \Leftrightarrow s(x)=0$ and for all $i \leqq l(x)(s(x \upharpoonright i) \geqq 0)$.*

Proof of Claim 1. The direction from left to right is trivial by induction on the construction of $x \in D$. To prove the other direction, assume the right-hand side is true. We use induction on the length of x. If for some $k < l(x)$, $s(x \upharpoonright k)=0$ then $x=(x \upharpoonright k)y$, for some y. Clearly, the induction hypothesis applies to $x \upharpoonright k$ and y. Consequently, both $x \upharpoonright k$, $y \in D$ and hence also $x \in D$. Otherwise, for all $k < l(x)$, $s(x \upharpoonright k) > 0$. Clearly, $x_{l(x)}=1$ (otherwise $s(x) > 0$). We also know that $x_1=0$. Hence, $x=0y1$, for some y. Clearly, this y satisfies the induction hypothesis stated in the right-hand side of Claim 1. Hence, $y \in D$ and consequently also $x \in D$.

As mentioned above, if n is odd the theorem is trivial. Hence, in all the proofs below we assume that n is even.

CLAIM 2. *For any $b \in \{0, 1\}$ and any $1 < i < n$ there exists a string $x \in D_n$ such that $x_i = b$.*

Proof of Claim 2. The proof is by induction on n. The claim is trivial if $n=2$. So assume $n > 2$. If $i=2$ then consider the strings $01y$, $0011z \in D_n$. If $i=n-1$, then consider the strings $y01$, $z0011 \in D_n$. Hence, without loss of generality, we can assume that $2 < i < n-1$. But then consider strings of the form $0y1$, where $y \in D_{n-2}$, and use the induction hypothesis.

CLAIM 3. *$\sigma \in S_n(D) \Rightarrow \sigma(1)=1$, $\sigma(n)=n$.*

Proof of Claim 3. Assume $\sigma(1)=i \neq 1$. Consider an $x \in D_n$ such that $x_i=1$ (use Claim 2). Then note that $x^\sigma = 1y \notin D_n$, for some string y, which is a contradiction. A similar proof shows that $\sigma(n)=n$.

CLAIM 4. *If $\sigma \in \mathrm{S}_n(D)$ and $\sigma[\{1, \cdots, i-1\}]=[\{1, \cdots, i-1\}]$ and $\sigma(i) < i$ then* (a) *i is even,* (b) *$\sigma(i)=i+1$,* (c) *$\sigma(i+1)=i$.*

Proof of Claim 4. To prove (a) assume on the contrary that i is odd. Consider an $x \in D_n$ such that $x = y0 \cdots 1z$, where $x_i=0$ and $x_{\sigma(i)}=1$ and $s(y)=0$. Applying σ to x we obtain that $x^\sigma = y^\sigma 1 \cdots$. But then $s(y^\sigma 1)=s(y^\sigma)-1=s(y)-1=-1 \ 0$. Hence, $x^\sigma \notin D_n$, by Claim 1, a contradiction.

To prove (b) assume on the contrary that $\sigma(i) > i+1$. For simplicity, assume that $\sigma(i)=i+2$ (a similar proof will work if $\sigma(i) \geqq i+2$). We distinguish several cases. If $\sigma(i+1)=i+1$ then consider the string $x=y0011 \cdots \in D_n$, with $l(y)=i-2$, $x_{i-1}=x_i=0$ and $x_{i+1}=x_{i+2}=1$. Then it is clear that $x^\sigma = y^\sigma 011 \cdots \notin D_n$, a contradiction. If $\sigma(i+1)=i+3$ then consider the string $x=y000111 \cdots \in D_n$, with $l(y)=i-2$, $x_{i-1}= x_i=x_{i+1}=0$ and $x_{i+2}=x_{i+3}=x_{i+4}=1$. Then it is clear that $x^\sigma = y^\sigma 011 \cdots \notin D_n$, a contradiction. If $\sigma(i+1) > i+3$ then consider the string $x=y0011 \cdots 1 \cdots \in D_n$, with $l(y)= i-2$, $x_{i-1}=x_i=0$ and $x_{i+1}=x_{i+2}=x_{\sigma(i+1)}=1$. Then it is clear that $x^\sigma = y^\sigma 011 \cdots \notin D_n$, a contradiction. Thus, we obtain a contradiction in all cases considered above. Hence, $\sigma(i)=i+1$. This completes the proof of (b).

To prove (c) use an argument similar to (b). Indeed, assume on the contrary, $\sigma(i+1) \neq i$. It follows that $\sigma(i+1) \geqq i+2$. If $\sigma(i+1)=i+2$ then take $x=y0011 \cdots \in D_n$, with $x_{i-1}=x_i=0$, $x_{i+1}=x_{i+2}=1$. If we apply σ to x then we obtain $x^\sigma = y^\sigma 011 \cdots \notin D_n$, which is a contradiction. If $\sigma(i+1)=i+3$ then take $x=y00101 \cdots \in D_n$, with $x_{i-1}=x_i=x_{i+2}=0$, $x_{i+1}=x_{i+3}=1$. If we apply σ to x then we obtain $x^\sigma = y^\sigma 011 \cdots \notin D_n$, which is a contradiction. In general, a similar proof works if $\sigma(i+1) \geqq i+3$. This completes the proof of (c).

Now we are ready to complete the proof of the theorem. Let $\sigma \in D_n$. We know that $\sigma(1)=1$. Let i_1 be minimal such that $\sigma(i_1) \neq i_1$ and for all $i < i_1(\sigma(i) < i_1)$. By minimality $\sigma(i_1) > i_1$. It follows from Claim 4 that i_1 is even and $\sigma(i_1)=i_1+1$ and $\sigma(i_1+1)=i_1$. Let i_2 be minimal i_1 such that $\sigma(i_2) \neq i_2$ and for all $i < i_2(\sigma(i) \ i_2)$. By minimality $\sigma(i_2) \ i_2$. Hence, Claim 4 applies again to show that i_2 is even and $\sigma(i_2)=i_2+1$

and $\sigma(i_2+1)=i_2$. Proceeding in this fashion we show that $\mathbf{S}_n(D)\subseteq\langle(i, i+1)$: $i\,n$ is even$\rangle$. It remains to show that, in fact, equality holds. Indeed, let $i<n$ be even. There are four possibilities for $x_i x_{i+1}$ in the string x:

$$X_1=y00\cdots,\qquad X_2=y01\cdots,\qquad X_3=y10\cdots,\qquad X_4=y11\cdots,$$

where y is a string of odd length. But then it is easy to see that for all $j=1,2,3,4$,

$$X_j\in D_n \Leftrightarrow X_j^{(i,i+1)}\in D_n,$$

which completes the proof of the theorem. □

3.2. Palindrome language. The palindrome language is defined as the set of all strings (in the alphabet Σ, with at least two elements) $u=u_1\cdots u_n$ such that for all i $(u_i=u_{n-i+1})$.

THEOREM 4. *If L is the palindrome then*

$$\sigma\in\mathbf{S}_n(L)\Leftrightarrow(\forall i\leqq n)(\sigma(n=n-\sigma(i)+1)).$$

Moreover, $\mathbf{S}_n(L)$ *is isomorphic to* $\mathbf{S}_{[n/2]}\times(\mathbf{Z}_2)^{[n/2]}$.

Proof. ($\Rightarrow$) Let $\sigma\in\mathbf{S}_n(L)$. Suppose that $\sigma(i)=j$. Consider the string $u=u_1\cdots u_n$ such that $u_j=u_{n-j+1}=0$, and $u_k=1$, for all $k\neq i$, $n-j-1$. Clearly, $u\in L_m$. Hence, also $u^\sigma\in L_n$. It follows that $u_{\sigma(i)}=u_j=0$ and consequently $u_{\sigma(n-i+1)}=0$. But this is true only if $\sigma(n-i+1)=n-j+1$, as desired.

($\Leftarrow$) This direction is obvious from the very definition of the palindrome.

To determine the group $\mathbf{S}_n(L)$, notice that by the previous result, a permutation $\sigma\in\mathbf{S}_n(L)$, is determined by the values $\sigma(1),\cdots,\sigma([n/2])$. Furthermore, note that if n is odd then $\sigma((n+1)/2)=(n+1)/2$. Now consider the permutation σ_0 such that for all $i\leqq n$, $\sigma_0(i)=n+1-i$ and put $G_n=\{\sigma\sigma_0\sigma\sigma_0^{-1}: \sigma\in\mathbf{S}_{[n/2]}\}$. It is easy to see that G_n is isomorphic to $\mathbf{S}_{[n/2]}$, moreover the group H_n generated by G_n and the transpositions $(i, n-i+1)$ is exactly the group

$$G_n\times(1, n)\times(2, n-1)\times\cdots\times([n/2], n-[n/2]-1).$$

Moreover, $H_n=\mathbf{S}_n(L)$. This completes the proof of the theorem. □

3.3. An algorithm for the invariance group of regular languages. Here we are interested in studying the complexity of membership in the invariance group of a regular language. To this end consider a term $t(x, y)$ built up from the variables x, y by concatenation. For example, $t(x, y)=xyx$, $t(x, y)=x^2yx^5y^3$, etc. are such terms. The number of occurrences of x and y in the term $t(x, y)$ is called the length of t and is denoted by $|t|$, e.g., $|t|=3$ and $|t|=11$, in the two previous examples. For any permutations σ, τ let the permutation $t(\sigma,\tau)$ be obtained from the term $t(x, y)$ by substituting each occurrence of x, y by σ,τ, respectively, and interpreting concatenation as the product of permutations. We know that the symmetry group $\mathbf{S}_n$ is generated by the cyclic permutation $c_n=(1,2,\cdots,n)$ and the transposition $\tau=(1,2)$ (in fact any transposition will do) [Wie64]. A sequence $\sigma=\langle\sigma_n: n\geqq 1\rangle$ of permutations is term-generated by the permutations c_n, τ if there is a term $t(x, y)$ such that for all $n\geqq 2$, $\sigma_n=t(c_n,\tau)$. We have the following theorem.

THEOREM 5. (1) *Let $\sigma=\langle\sigma_n: n\geqq 1\rangle$ be a sequence of permutations which is term-generated by the permutations $c_n=(1,2,\cdots,n)$, $\tau=(1,2)$. Then for any regular language L, L^σ is also regular.*

(2) *For any term t of length $|t|$ the problem of testing whether, for a regular language L, $L=L^\sigma$, where $\sigma=\langle\sigma_n: n\geqq 1\rangle$ is a sequence of permutations generated by the term t via the permutations $c_n=(1,2,\cdots,n)$, $\tau=(1,2)$, is decidable; in fact it has complexity $O(2^{|t|})$.*

Proof. Part (2) is an immediate consequence of the proof of part (1) and the solvability of the equality problem for regular languages [Harr 78]. So we concentrate only on the proof of (1). To prove the theorem we need the following claim, whose proof is easy and left to the reader.

CLAIM.

$$L \in \mathbf{REG} \Rightarrow \{x: 0x \in L\} \in \mathbf{REG}.$$

$$L \in \mathbf{REG} \Rightarrow \{x: x1 \in L\} \in \mathbf{REG}.$$

$$L \in \mathbf{REG} \Rightarrow \{x: 0x1 \in L\} \in \mathbf{REG}.$$

$$L \in \mathbf{REG} \Rightarrow \{x: 1x0 \in L\} \in \mathbf{REG}.$$

First we show how to prove the theorem when $\sigma_n = (1, n)$. Indeed,

$$L_n^{(1,n)} = \{x \in 2^n: x_n x_2 \cdots x_{n-1} x_1 \in L\}$$

and this last set is the union of the following four sets:

$$\{x \in 2^n: 0x_2 \cdots x_{n-1}0 \in L\}, \qquad \{x \in 2^n: 1x_2 \cdots x_{n-1}1 \in L\},$$

$$\{x \in 2^n: 0x_2 \cdots x_{n-1}1 \in L\}, \qquad \{x \in 2^n: 1x_2 \cdots x_{n-1}0 \in L\}.$$

This completes the proof in view of the above claim. A similar proof will yield the result when each $\sigma_n = (1, 2)$. Next we use the above result for the transpositions $(1, n)$ to prove the result for the n-cycles, $\sigma_n = c_n$. Indeed,

$$\begin{aligned} L \in \mathbf{REG} &\Rightarrow \{x_1 \cdots x_n: x1 \in L\} \in \mathbf{REG} \\ &\Rightarrow \{x_1 \cdots x_n: x_1 \cdots x_n 1 \in L\} \in \mathbf{REG} \\ &\Rightarrow \{x_1 \cdots x_n: 1x_2 \cdots x_n x_1 \in L\} \in \mathbf{REG} \\ &\Rightarrow \{x_1 \cdots x_n: x_2 \cdots x_n x_1 \in L\} \in \mathbf{REG}. \end{aligned}$$

Finally, the theorem follows by using the following product formula, which is valid for any permutations $\tau_1, \tau_2 \in \mathbf{S}_n$,

$$L_n^{\tau_1 \tau_2} = (L_n^{\tau_1})^{\tau_2}.$$

This completes the proof of the theorem. □

The assumption on term generation of the sequence $\langle \sigma_n: n \geqq 1 \rangle$ of permutations, made in the last theorem, is necessary as the following example shows.

Example 6. Let R be a recursively enumerable but nonrecursive set. Consider the permutation σ_n, which is equal to $(1, n)$, if $n \in R$, and is equal to id_n, if $n \notin R$, where id_n is the identity permutation on n letters. Consider the regular language defined by $L = 10^*$. Then it is easy to see that $L_n^\sigma = \{10^n: n+1 \notin R\} \cup \{0^n 1: n+1 \in R\}$. It follows that $n \in R \Leftrightarrow 0^{n-1}1 \in L^\sigma$. Hence, L^σ is not even a recursive language, although L is regular.

4. Constructing languages with given invariance groups. This section is concerned with the problem of realizing specific sequences of finite permutation groups by languages $L \subseteq \{0, 1\}^*$. A language L is said to realize a sequence $\mathbf{G} = \langle G_n: n \geqq 1 \rangle$ of permutation groups $G_n \leqq \mathbf{S}_n$ if it is true that $\mathbf{S}_n(L) = G_n$, for all n. We consider the following types of groups and determine regular as well as nonregular languages realizing them.

Reflection. $R_n = \langle \rho \rangle$, where $\rho(i) = n+1-i$ is the reflection permutation,

Cyclic. $C_n = \langle (1, 2, \cdots, n) \rangle$.

Dihedral. $D_n = C_n \times R_n$.

Hyperoctahedral. $O_n = \langle (i, i+1)$: i is even $\leqq n \rangle$.

THEOREM 7. (1) *Each of the identity, reflection, cyclic (for $n \neq 3, 4, 5$), dihedral, and hyperoctahedral groups can be realized by regular languages.*

(2) *Each of the identity, cyclic, and dihedral groups can be realized by languages L such that $L \notin SIZE(n^{O(1)})$.*

Proof. (1) For each of the above-mentioned types of groups we provide a regular language realizing it.

Identity. This case is simple: take $L = 0^*1_*$.

Dihedral. Let $L = 0^*1^*0^* \cup 1^*0^*1^*$. It is clear that $D_n \subseteq \mathbf{S}_n(L)$. Let ρ be the reflection permutation defined by $\rho(i) = n+1-i$ and let $\sigma = (1, 2, \cdots, n)$. It is easy to check that $\sigma\rho\sigma = \rho$. It follows that $D_n = \{\sigma^k \rho^l: k \leqq n,\ l = 0, 1\}$. Next we prove the following claim.

CLAIM. *For all $\tau \in \mathbf{S}_n$, if addition is modulo n,*

$$\tau \in D_n \Leftrightarrow \forall i \leqq n(\tau(i+1) = \tau(i) + 1)$$

or

$$\forall i \leqq n(\tau(i) = \tau(i+1) + 1).$$

Proof of the claim. From left to right the equivalence is easily verified for the permutations $\sigma^k \rho^l$ $(1 \leqq k \leqq n,\ l = 0, 1)$. For example, $\sigma(i+1) = \sigma(i) + 1$ and $\rho(i) = \rho(i+1) + 1$. To prove the other direction, assume that τ satisfies the right-hand side. Say, $\tau(1) = k$. It is then easy to see that either $\tau = \sigma^{k-1}$ or $\tau = \sigma^k \rho$. This completes the proof of the claim.

It remains to show that $\mathbf{S}_n(L) \subseteq D_n$. If $n \leqq 3$ the result is trivial. So assume that $n \geqq 4$. Let $\tau \notin D_n$. There exists an $i \leqq n-1$ such that $|\tau(i+1) - \tau(i)| \geqq 2$. Let us suppose that $1 \leqq \tau(i) + 1 < \tau(i+1) \leqq n$. Then we have that

$$x = 0^{i-1} 1^2 0^{n+1-i} \in L_n, \qquad x^\tau = 0^{\tau(i)-1} 1 0^{\tau(i+1)-1} 1^{n-\tau(i+1)} \notin L_n.$$

Reflection. Let $L = 0^*1^*0^*$. It is clear that $R_n \subseteq \mathbf{S}_n(L)$. We want to show that $\mathbf{S}_n(L) \subseteq R_n$. By the proof given in the case of dihedral groups we have that $\mathbf{S}_n(L_n) \subseteq D_n$. Assume on the contrary that $\tau \in \mathbf{S}_n(L)$, but $\tau \in D_n - R_n$. It follows that $\tau = \sigma^i \rho$, for some $i \geqq 1$. Since $\rho \in \mathbf{S}_n(L)$ we obtain that $\sigma^i \in \mathbf{S}_n(L)$, which is a contradiction.

Cyclic. First assume that $n = 2$. Then consider the regular language

$$L = (01 \cup 10)0^*1^*$$

and notice that $\mathbf{S}_n(L) = (1, 2)$.

Next assume that $n \geqq 6$. Consider the regular language $L = L^1 \cap L^2$ where L^1 is the language

$$1^*0^*1^* \cup 0^*1^*0^* \cup 101000^*1 \cup 0^*1101000^* \cup 0^*011010$$
$$\cup\, 0^*001101 \cup 10^*00110 \cup 010^*0011$$

and L^2 is the language

$$\overline{10^*00101}.$$

Clearly, $C_n \subseteq \mathbf{S}_n(L)$. In view of the result on dihedral groups we have that $\mathbf{S}_n(L) \subseteq D_n$. Let $x = 101000^{n-6}1 \in L_n$. Then $x^\rho = 10^{n-6}00101 \notin L_n$, where $\rho(i) = n+1-i$. Hence, $C_n = \mathbf{S}_n(L)$, for $n \geqq 6$.

It is interesting to note that for $3 \leqq n \leqq 5$ the groups C_n are not representable. This is obvious for $n=3$, since $C_3 = \mathbf{A}_3$. For $n = 4, 5$, one can show directly that for any boolean function $f \in \mathbf{B}_n$, if $C_n \subseteq \mathbf{S}(f) \subseteq D_n$ then $\mathbf{S}(f) = D_n$.

Hyperoctahedral. Consider the language L consisting of the set of all finite strings $x = (x_1, \cdots, x_k)$ such that for some $i \leqq k/2$, $x_{2i-1} = x_{2i}$. The regularity of the language follows from the obvious equality

$$L = (\Sigma\Sigma)^*(00 \cup 11)\Sigma^*.$$

For any set $I = \{i, j\}$ of indices, let f_I be the n-ary boolean function defined by

$$f_I(x) = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{if } x_i \neq x_j. \end{cases}$$

Put $m = [n/2]$. For each $i = 1, \cdots, m$ consider the two-element sets $I_i = \{2i-1, 2i\}$ and the functions f_{I_i} defined above. Consider the boolean function

$$f = f_{I_1} \vee \cdots \vee f_{I_m}.$$

It is then clear that $\mathbf{S}_n(L) = \mathbf{S}(f)$. It is also easy to see that this last group consists of all permutations $\sigma \in \mathbf{S}_n$ which permute the blocks I_i, $i = 1, \cdots, m$. In fact this last group has exactly $2^{[n/2]} \cdot [n/2]!$ elements.

To prove part (2) of the theorem we use Lupanov's theorem (see § 2.3), i.e.,

$$|\{f \in \mathbf{B}_n \colon c(f) < q\}| = O(q^{q+1}).$$

Identity. By Lupanov's theorem we have that

$$|\{f \in \mathbf{B}_n \colon c(f) \leqq n^{\log n}\}| = 2^{O(n^{\log n}(\log n)^2)} \ll 2^{2^n}$$
$$\sim |\{f \in \mathbf{B}_n \colon S(f) = 1\}|.$$

It follows that for all but a finite number of n there exists $f_n \in \mathbf{B}_n$ such that $L(f_n) \geqq n^{\log n}$ and $\mathbf{S}(f_n) = 1$. If we define a language L such that for all n, $L_n = f_n$, then the proof is complete.

Cyclic. The result will follow by a proof similar to the above if we could prove that

$$(3) \qquad |\{f \in \mathbf{B}_n \colon S(f) = D_n\}| \geqq 2^{2^n/n - n(n-1)/2} \gg 2^{O(n^{\log n}/n(\log n)^2)}.$$

Indeed, the left part of the above inequality is true because one may independently assign a value of 0, 1 to each orbit, except for orbits of words having 2 or 3 occurrences of the symbol 1. Let $\sigma = (1, 2, \cdots, n)$ be the n-cycle and let ρ be the reflection on n letters. We agree to have $f(v) \neq f(w)$, where $|v|_1 = |w|_1 = 2$ and

$$v \in \{(1^2 0^{n-2})^{\sigma^i} \colon 0 \leqq i \leqq n-1\}, \qquad w \in 2^n - \{(1^2 0^{n-2})^{\sigma^i} \colon 0 \leqq i \leqq n-1\}.$$

This removes n. Choose 2 independent choices while adding one choice of 0 or 1. We agree to have $f(v) \neq f(w)$, where $|v|_1 = |w|_1 = 3$ and

$$v \in \{(101000^{n-6}1)^{\sigma^i} \colon 0 \leqq i \leqq n-1\}, \qquad w \in \{(10^{n-6}00101)^{\sigma^i} \colon 0 \leqq i \leqq n-1\}.$$

Again, this removes n. Choose 2 independent choices while adding one choice of 0 or 1. Hence, the proof of the desired lower bound (1) is complete.

Dihedral. By [Ber71, p. 171], $\Theta(D_n) \geqq 2^{n-1}/n$. An argument similar to the one for cyclic groups used above shows that

$$|\{f \in \mathbf{B}_n : \mathbf{S}(f) = D_n\}| \geqq 2^{2^{n-1}/n - n(n-1)/2} \gg 2^{O(n^{\log n}/n(\log n)^2)}.$$

This completes the proof of the theorem. □

There is another interesting way for realizing the cyclic groups C_n, for $n \geqq 4$. For any groups G, H, put $[G, H] = \{g^{-1}h^{-1}gh : g \in G, h \in H\}$. Let $G, H \leqq \mathbf{S}_n$ be two permutation groups. Consider the set of words in G^* defined by

$$L_{G,H} = \{w \in G^* : w \in H\}.$$

(The reader should be warned of the different interpretation of w in the expressions $w \in G^*$ and $w \in H$; the former is a word in G^* and the latter is an element of a group.)

THEOREM 8. *For any permutation groups $G, H \leqq \mathbf{S}_n$, if $[G, G]$ is not a subset of the normal subgroup H of G, then $\mathbf{S}_n(L_{G,H}) = C_n$, for $n \geqq 4$.*

Proof. First we show that $C_n \subseteq \mathbf{S}_n^+(L_{G,H})$. Indeed, consider the cyclic permutation $c_n = (1, 2, \cdots, n)$ and notice that for $w = \sigma_1 \cdots \sigma_n \in G^*$,

$$w^{c_n} = \sigma_{c_n(1)} \cdots \sigma_{c_n(n)} = \sigma_2\sigma_3 \cdots \sigma_n\sigma_1 = \sigma_1^{-1} w \sigma_n.$$

It follows from the normality of H in G that $c_n \in \mathbf{S}_n^+(L_{G,H})$. This completes the proof of $C_n \subseteq \mathbf{S}_n^+(L_{G,H})$. Next we prove that $\mathbf{S}_n(L_{G,H}) \subseteq C_n$. Indeed, let ρ be a permutation in $\mathbf{S}_n - D_n$. It follows from the proof of Theorem 7 that either (A) there exists an i such that $|\rho(i+1) - \rho(i)| \bmod n > 1$, or (B) $|\rho(n) - \rho(1)| \bmod n > 1$. We show that $\rho \notin \mathbf{S}_n(L_{G,H})$. First we consider case (A) and distinguish four subcases.

Case 1. $1 \leqq \rho(i) < \rho(i+1)n$.

Let σ, τ be given such that $[\sigma, \tau] = \sigma\tau\sigma^{-1}\tau^{-1} \notin H$. Let $j = \rho^{-1}(\rho(i)+1)$, $k = \rho^{-1}(\rho(i+1)+1)$. Consider $w = \sigma_1 \cdots \sigma_n \in G^n$, where $\sigma_i = \sigma$, $\sigma_{i+1} = \sigma^{-1}$, $\sigma_j = \tau$, $\sigma_k = \tau^{-1}$, and all other σ_l's are equal to 1. Then we have that $w = \sigma\sigma^{-1}\tau\tau^{-1}$ or $\sigma\sigma^{-1}\tau^{-1}\tau$ depending, respectively, on whether or not $j < k$ or $k < j$. In either case $w = 1$, but $w^\rho = \sigma\tau\sigma^{-1}\tau^{-1} \notin H$.

Case 2. $\rho(i) < \rho(i+1) \leqq n$.

Let σ, τ be given such that $[\sigma, \tau] = \sigma\tau\sigma^{-1}\tau^{-1} \notin H$. Let $j = \rho^{-1}(\rho(i)-1)$ and $k = \rho^{-1}(\rho(i)+1)$. Choose w such that $w = \sigma_1 \cdots \sigma_n \in G^n$, where $\sigma_j = \sigma$, $\sigma_{i+1} = \tau^{-1}$, $\sigma_i = \tau$, $\sigma_k = \sigma^{-1}$ and all other σ_l's are equal to 1. Then it is clear that $w = 1$, while $w^\rho \notin H$.

Case 3. $1 \leqq \rho(i+1) < \rho(i) < n$. This is similar to case 1.

Case 4. $1 < \rho(i+1) < \rho(i) \leqq n$. This is similar to case 1.

Case (B) is handled exactly as before. Hence, we have proved that $\mathbf{S}_n(L_{G,H}) \subseteq D_n$. It remains to show that in fact $\mathbf{S}_n(L_{G,H}) = C_n$. Since $[G, G]$ is not a subset of H, G/H cannot be abelian. Therefore, there exist elements $g_1, g_2, g_3, g_4 \in G$ such that

$$g_1g_2g_3g_4 \in H, \quad \text{but} \quad g_4g_3g_2g_1 \notin H.$$

It follows that the reflection permutation does not belong to $\mathbf{S}_n(L_{G,H})$, which completes the proof of the theorem. □

Given a language $L \subseteq \Sigma^*$ over the alphabet Σ the syntactic semigroup G_L of L is defined as follows. Define $w \equiv w' \bmod L$ if for all $u, v \in \Sigma^*$, $uwv \in L \Leftrightarrow uw'v \in L$. Then let G_L be the quotient of Σ^* modulo the equivalence relation $\equiv \bmod L$. Recall that the Krohn–Rhodes theorem [Arb69] states that the syntactic semigroup G_L of any given regular language L is the homomorphic image of a wreath product of cyclic simple groups, noncyclic simple groups, and three particular nongroup semigroups called "units." If G is abelian and $H = 1$, then it is clear that $\mathbf{S}_n(L_{G,H}) = \mathbf{S}_n$. If G is a nonabelian group and $H = 1$, then Theorem 8 yields that $\mathbf{S}_n(L_{G,H}) = C_n$. We have seen families of these groups as invariance groups of regular languages. However, we have

examples of representable groups whose homomorphic image is not representable, (e.g., $(1,2,3)$ is the homomorphic image of $(1,2,3)(4,5,6)$), thus indicating that it is unlikely that the Krohn–Rhodes theorem can be used to characterize those families of invariance groups of regular languages. Similarly, from the examples given in the paper, there is no invariance group structure preserved when taking regular operations: from $\mathbf{S}_n(L)$ and $\mathbf{S}_n(L')$, we cannot say anything in general about $\mathbf{S}_n(M)$, where $M = L \# L'$ and $\#$ is a boolean operation or language concatenation or where $M = L^*$ (Kleene star). This blocks a natural attempt to inductively define the families of invariance groups of regular languages.

It is not known whether there is a characterization of those sequences of groups which can be realized by regular languages. However, it is interesting to note that for regular languages L the invariance group $\mathbf{S}_{2n}(L)$ can never be equal to the $\{1, 2, \cdots, n\}$ point-stabilizer of $\mathbf{S}_{2n}$.

THEOREM 9. (1) *There is no regular language L such that for all but a finite number of n we have that*

$$\mathbf{S}_{2n}(L) = (\mathbf{S}_{2n})_{\{1,2,\cdots,n\}}.$$

(2) *There is a regular language L such that for all n we have that*

$$\mathbf{S}_{2n}(L) = (\mathbf{S}_{2n})_{\{2i:\, i \leqq n/2\}}.$$

Proof. (1) By the pumping lemma for regular languages [Harr78] there exist words $a_i, b_i, i < m$ and $\bar{a}_j, \bar{b}_j, j < \bar{m}$ and languages $L_i, \bar{L}_j$ such that

$$L = \bigcup_{im} a_i b_i^* L_i, \qquad \bar{L} = \bigcup_{i\bar{m}} \bar{a}_j \bar{b}_j^* \bar{L}_j,$$

where $\neg L = \{0,1\}^* - L$ is the complement of L. Let r be the least common multiple of the lengths of all the above words. Put $i = r+1$, $j = i + r$, and $n_0 = 3r$. Consider the transposition $\tau = (i, j)$ and let $n \geqq n_0$. Then for any word w of length n we consider the following two cases.

Case 1. $w \in L_n$.

Then for some $i_0 < m$ and some s we have that w must be of the form $a_{i_0} b_{i_0}^s c_{i_0}$. The ith position in the word w falls within the block b_{i_0}. Since the length of b_{i_0} divides r the jth position of the word w falls in exactly the same position with respect to the block b_{i_0}. It follows that $w_i = w_j$ and hence $w^\tau = w$.

Case 2. $w \notin L_n$.

This is similar to the proof of Case 1.

It follows from the above that $\tau \in \mathbf{S}_n(L)$, as desired. This completes the proof of part (1).

(2) Consider the languages $L' = 0^*$ and $L'' = 1^*0^*$. It is clear that for all n, $\mathbf{S}_n(L') = \mathbf{S}_n$, and $\mathbf{S}_n(L'') = 1$. Let L be the set of all words w of even length $2n$ such that

$$w_1 w_3 \cdots w_{2n-1} \in L', \qquad w_2 w_4 \cdots w_{2n} \in L''.$$

Clearly, L is a regular language and $\mathbf{S}_{2n}(L) \supseteq (\mathbf{S}_{2n})_{\{2i:\, i\leqq n/2\}}$. It remains to show that in fact $\mathbf{S}_{2n}(L) \subseteq (\mathbf{S}_{2n})_{\{2i:\, i\leqq n/2\}}$. Indeed, let $\sigma \in \mathbf{S}_{2n}(L)$ and decompose σ as a product of the disjoint cycles $\sigma_1 \cdots \sigma_k$. Assume on the contrary that there exists an i_0 such that $\sigma_{i_0} = (a_1, \cdots, a_r)$ and

(i) either there exists a $1 \leqq j_0 < r$ such that a_{j_0} is even and a_{j_0+1} is odd,

(ii) or a_r is even and a_1 is odd.

We treat only case (ii), the other case being entirely similar. Consider a word w defined as follows. Let $w_1 = w_3 = \cdots = w_{2n-1} = 0$ and $w_2 = w_4 = \cdots = w_{a_r} = 1$ and the

remaining w_i's equal to 0. Then $w \in L$. However, $(w^\sigma)_{a_1} = 1$, where a_1 is odd, and so

$$(w^\sigma)_1(w^\sigma)_3 \cdots (w^\sigma)_{2n-1} \notin L'.$$

It follows that $w^\sigma \notin L$. Hence, $\sigma \notin \mathbf{S}_{2n}(L)$, a contradiction. □

5. Representations of permutation groups. The aim of this section is to give general results on permutation groups $G \leqq \mathbf{S}_n$ which can be represented as the invariance groups of boolean functions, i.e., $G = \mathbf{S}(f)$ for some $f \in \mathbf{B}_n$. It will be seen in the sequel that there is a rich class of permutation groups which are representable in this way.

The main motivation for the results of the present section is the simple observation that the alternating group $\mathbf{A}_n$ is not the invariance group of any boolean function $f \in \mathbf{B}_n$, provided that $n \geqq 3$. Although this will follow directly from our representation theorem it will be instructive to give a direct proof. Suppose that the invariance group of $f \in \mathbf{B}_n$ contains $\mathbf{A}_n$. Given $x \in 2^n$, for $3 \leqq n$, there exist $1 \leqq i < j \leqq n$ such that $x_i = x_j$. It follows that the alternating group $\mathbf{A}_n$ and a transposition fix f on x, and hence $\mathbf{S}_n$ does as well. As this holds for every $x \in 2^n$, it follows that $\mathbf{S}(f) = \mathbf{S}_n$. In fact it is clear, using part (1) of Theorem 1, that $\mathbf{A}_n$ is not isomorphic to the invariance group $\mathbf{S}(f)$ of any $f \in \mathbf{B}_n$. However, $\mathbf{A}_n$ is isomorphic to the invariance group $\mathbf{S}(f)$ for some boolean function $f \in \mathbf{B}_{n(\log n+1)}$ (see Theorem 11 below).

One can generalize the notion of invariance group for any language $L \subseteq \{0, 1, \cdots, k\}^*$ by setting $L_n = L \cap \{0, \cdots, k\}^n$ and $\mathbf{S}(L_n)$ to be

$$\{\sigma \in \mathbf{S}_n : \forall x_1, \cdots, x_n \in \{0, 1, \cdots, k\}(x_1, \cdots, x_k \in L_n \Leftrightarrow x_{\sigma(1)}, \cdots, x_{\sigma(n)} \in L_n)\}.$$

We leave the details of the proof of the following fact as an exercise for the reader.

FACT. *For all n, there exist groups $G_n \leqq \mathbf{S}_n$ which are strongly representable as $G_n = \mathbf{S}(L_n)$ for some $L \subseteq \{0, 1, \cdots, n-1\}^n$ but which are not so representable for any language $L' \subseteq \{0, 1, \cdots, n-2\}^n$.*

Proof. The alternating group $\mathbf{A}_n = \mathbf{S}(L_n)$, where $L_n = \{w \in \{0, \cdots, n-1\}^n : \sigma_w \in \mathbf{A}_n\}$, where $\sigma_w : i \mapsto w(i-1)+1$. By a variant of the previous argument, $\mathbf{A}_n$ is not so representable by any language $L' \subseteq \{0, 1, \cdots, n-2\}^n$. □

Compared to the difficulties regarding the question of representing permutation groups $G \leqq \mathbf{S}_n$ in the form $G = \mathbf{S}(f)$, for some $f \in \mathbf{B}_n$, it is interesting to note that a similar representation theorem for the groups $\mathbf{S}(x) = \{\sigma \in \mathbf{S}_n : x^\sigma = x\}$, where $x \in 2^n$, is relatively easy. It turns out that these last groups are exactly the permutation groups which are isomorphic to $\mathbf{S}_k \times \mathbf{S}_{n-k}$ for some k. Indeed, given $x \in 2^n$ let

$$X = \{i : 1 \leqq i \leqq n \text{ and } x_i = 0\}, \qquad Y = \{i : 1 \leqq i \leqq n \text{ and } x_i = 1\}.$$

It is then easy to see that $\mathbf{S}(x)$ is isomorphic to $\mathbf{S}_X \times \mathbf{S}_Y$. In fact, $\sigma \in \mathbf{S}(x)$ if and only if $X^\sigma = X$ and $Y^\sigma = Y$.

5.1. Elementary properties. Before we proceed with the general results we will prove several simple observations that will be used frequently in the sequel. We begin with a few useful definitions. For any $f \in \mathbf{B}_n$, let $\mathbf{S}^+(f) = \{\sigma \in \mathbf{S}_n :$ for all $x \in 2^n(f(x) = 0 \Rightarrow f(x^\sigma) = 0)\}$. For any permutation group $G \leqq \mathbf{S}_n$ and any $\Delta \subseteq \{1, 2, \cdots, n\}$ let G_Δ be the set of permutations $\sigma \in G$ such that (for all $i \in \Delta)(\sigma(i) = i)$. G_Δ is called the pointwise stabilizer of G on Δ. Notice that $(\mathbf{S}_n)_{\{k+1, \cdots, n\}} = S_k$, for $k \leqq n$. For any permutation σ and permutation group G let $G^\sigma = \sigma^{-1} G \sigma$, also called a conjugate of G by σ. For any $f \in \mathbf{B}_n$ let $1 \oplus f \in \mathbf{B}_n$ be defined by $(1 \oplus f)(x) = 1 \oplus f(x)$, for $x \in 2^n$. If $f_1, \cdots, f_k \in \mathbf{B}_n$ and $f \in \mathbf{B}_k$ then $g = f(f_1, \cdots, f_k) \in \mathbf{B}_n$ is defined by $g(x) = f(f_1(x), \cdots, f_k(x))$. The following theorem contains several useful observations that will be used frequently in the sequel.

THEOREM 10. (1) *If* $f \in \mathbf{B}_n$ *is symmetric then* $\mathbf{S}(f) = \mathbf{S}_n$.

(2) $\mathbf{S}(f) = \mathbf{S}(1 \oplus f)$, *for all* $f \in \mathbf{B}_n$.

(3) *For any permutation* σ, $\mathbf{S}(f^\sigma) = \mathbf{S}(f)^\sigma$.

(4) *For each* $f \in \mathbf{B}_n$, $\mathbf{S}(f) = \mathbf{S}^+(f)$.

(5) *If* $f_1, \cdots, f_k \in \mathbf{B}_n$ *and* $f \in \mathbf{B}_k$ *and* $g = f(f_1, \cdots, f_k) \in \mathbf{B}_n$ *then* $\mathbf{S}(f_1) \cap \cdots \cap \mathbf{S}(f_k) \subseteq \mathbf{S}(g)$.

(6) *(For all* $k \leqq n)(\exists f \in \mathbf{B}_n)\mathbf{S}(f) = \mathbf{S}_k$.

Proof. The proofs of (1)-(3), (5) are easy and are left as an exercise to the reader. To prove (4), notice that $\mathbf{S}^+(f)$ is a group and trivially $\mathbf{S}(f) \subseteq \mathbf{S}^+(f)$. Now let $\sigma \in \mathbf{S}^+(f)$ and suppose that $f(x^\sigma) = 0$ holds. Since, $\sigma^{-1} \in \mathbf{S}^+(f)$ we have that $f(x) = f((x^\sigma)^{\sigma^{-1}}) = 0$. It follows that $\mathbf{S}^+(f) \subseteq \mathbf{S}(f)$, as desired. To prove (6) we consider two cases. If $k+2 \leqq n$, define f by

$$f(x) = \begin{cases} 1 & \textit{if } x_{k+1} \leqq x_{k+2} \leqq \cdots \leqq x_n \\ 0 & \text{otherwise.} \end{cases}$$

Let $\sigma \in \mathbf{S}(f)$. First notice that for all $i > k(\sigma(i) > k)$. Next, it is easy to show that if σ is a nontrivial permutation then there can be no $k \leqq i < j \leqq n$ such that $\sigma(j) < \sigma(i)$. This proves the desired result. If $k = n-1$, then the function f must be defined as follows.

$$f(x) = \begin{cases} 1 & \textit{if } x_1, \cdots, x_{n-1} \leqq x_n \\ 0 & \text{otherwise.} \end{cases}$$

A similar proof will show that $\mathbf{S}(f) = \mathbf{S}_{n-1}$. This completes the proof of the theorem. □

We define a permutation group $G \leqq \mathbf{S}_n$ to be *representable* (respectively, *strongly representable*) if there exists an integer k and a function $f \in \mathbf{B}_{n,k}$ (respectively, with $k = 2$) such that $G = \mathbf{S}(f)$. $G \leqq \mathbf{S}_n$ is called *weakly representable* if there exists an integer k, an integer $m < n$, and a function $f: m^n \to k$ such that $G = \mathbf{S}(f)$. It will be seen in the sequel (representability theorem) that the distinction between representable and strongly representable is superfluous since these two notions coincide.

Notice the importance of assuming $m < n$ in the above definition of weak representability. If $m = n$ were allowed, then every permutation group would be weakly representable. Indeed, given any permutation group $G \leqq \mathbf{S}_n$ define the function f as follows:

$$f(x_1, \cdots, x_n) = \begin{cases} 0 & \textit{if } (x_1, \cdots, x_n) \in G \\ 1 & \text{otherwise} \end{cases}$$

(here, we think of $(x_1, \cdots, x_n)$ as the function $i \to x_i$ in n^n) and notice that for all $\sigma \in \mathbf{S}_n$, $\sigma \in \mathbf{S}(f)$ if and only if for all $\tau \in \mathbf{S}_n$ $(\tau \in G \Leftrightarrow \tau\sigma \in G)$. Hence $G = \mathbf{S}(f)$, as desired.

Another issue concerns the number of variables allowed in a boolean function in order to represent a permutation group $G \leqq \mathbf{S}_n$. We can also consider representing functions by using additional variables, but as the following theorem shows, every group becomes representable if enough variables are allowed.

THEOREM 11 (Isomorphism Theorem). *Every finite permutation group* $G \leqq \mathbf{S}_n$ *is isomorphic to the invariance group of a boolean function* $f \in \mathbf{B}_{n(\log n+1)}$.

Proof. First, let us give some notation. Let w be a word in $\{0, 1\}^*$. $|w|_1$ is the number of occurrences of 1 in w, and w_i is the ith symbol in w, where $1 \leqq i \leqq |w| =$ length of w. The word w is monotone if for all $1 \leqq i < j \leqq |w|$, $w_i = 1 \Rightarrow w_j = 1$. The complement of w, denoted by $\bar{w}$ is the word which is obtained from w by "flipping" each bit w_i,

i.e., $|w|=|\bar{w}|$ and $\bar{w}_i = 1 \oplus w_i$, for all $1 \leqq i \leqq |w|$. Fix n and let $s = \log n + 1$. View each word $w \in \{0,1\}^{ns}$ (of length ns) as consisting of n-many blocks, each of length s, and let $w(i) = w_{(i-1)s+1} \cdots w_{is}$ denote the ith such block. For a given permutation group $G \leqq \mathbf{S}_n$ let L_G be the set of all words $w \in \{0,1\}^{ns}$ such that

- either (i) $|w|_1 = s$, and if the word w is divided into n-many blocks $w(1), w(2), \cdots, w(n)$, each of length s, then exactly one of these blocks consists of 1's, while the rest of the blocks consist only of 0's,
- or (ii) $|w|_1 \leqq s-1$ and for each $1 \leqq i \leqq n$, the complement $\bar{w}$ of the ith block of w is monotone (this implies that each $w(i)$ consists of a sequence of 1's concatenated with a sequence of 0's),
- or (iii) $|w|_1 \geqq n$ and for each $1 \leqq i \leqq n$, $w(i)_1 = 0$ (i.e., the first bit of $w(i)$ is 0) and the binary representations of the words $w(i)$, say $bin(w,i)$, are mutually distinct integers and $\sigma_w \in G$, where $\sigma_w: \{1, \cdots, n\} \to \{1, \cdots, n\}$ is the permutation defined by

$$\sigma_w(i) = bin(w,i).$$

The intuition for items (i) and (ii) above is the following. The words with exactly s-many 1's have all these 1's in exactly one block. This guarantees that any permutation "respecting" the language L_G must map blocks to blocks. By considering words with a single 1 (which by monotonicity must be located at the first position of a block) we guarantee that each permutation "respecting" L_G must map the first bit of a block to the first bit of some other block. Inductively, by considering the word with exactly $(r-1)$-many 1's, all located at the beginning of a single block, while all other bits of the word are 0's, we guarantee that each permutation "respecting" L_G must map the $(r-1)$st bit of each block to the $(r-1)$st bit of some other block. It follows that any permutation respecting L_G must respect blocks as well as the order of elements in the blocks; i.e., for every permutation $\tau \in \mathbf{S}_{ns}(L_G)$,

$$(\forall 0 \leqq k < n)(\exists 0 \leqq m < n)(\forall 1 \leqq i \leqq n)\tau(ks+i) = ms+i.$$

Call such a permutation "s-block invariant." Given a permutation $\tau \in \mathbf{S}_{ns}(L_G)$ let $\bar{\tau} \in \mathbf{S}_n$ be the induced permutation defined by

$$\bar{\tau}(k) = m \Leftrightarrow (\forall 1 \leqq i \leqq n)\tau(ks+i) = ms+i.$$

We claim that $G = \{\bar{\tau}: \tau \in \mathbf{S}^+_{ns}(L_G)\}$. Indeed, to prove ($\subseteq$) notice that every element $\bar{\tau}$ of G gives rise to a unique "s-block invariant" permutation τ. If $w \in L_G$ and $|w|_1 \leqq s$, then by s-block invariance of τ, $w^\tau \in L_G$. This proves ($\subseteq$). If $w \in L_G$ and $\sigma_w \in G$, then $\sigma_{(w^\tau)} = \sigma_w \bar{\tau} \in G$ (composition is from the right). To prove ($\supseteq$) let $w \in L_G$ be such that σ_w is the identity on $\mathbf{S}_n$. Then for any $\tau \in \mathbf{S}_{ns}(L_G)$, $w^\tau \in L_G$, so $\sigma_{(w^\tau)} = \sigma_w \bar{\tau} = \bar{\tau} \in G$, which proves the above claim. This completes the proof of the theorem. □

Clearly, the idea of the proof of the previous theorem can also be used to show that for any alphabet Σ, if $L \subseteq \Sigma^n$, then $\mathbf{S}_n(L)$ (the set of permutations in $\mathbf{S}_n$ "respecting" the language L) is isomorphic to $\mathbf{S}_{ns}(L')$, for some $L' \subseteq \{0,1\}^{ns}$, where $s = 1 + \log|\Sigma|$.

We conclude by comparing the different definitions of representability given above.

THEOREM 12. *For any permutation group $G \leqq \mathbf{S}_n$ the following statements are equivalent*:

(1) *G is representable.*

(2) *G is the intersection of a finite family of strongly representable permutation groups.*

(3) *For some m, G is a pointwise stabilizer of a strongly representable group over $\mathbf{S}_{n+m}$, i.e., $G = (\mathbf{S}_{n+m}(f))_{\{n+1,\cdots,n+m\}}$, for some $f \in \mathbf{B}_{n+m}$ and $m \leqq n$.*

Proof. First we prove that (1)⇒(2). Indeed, let $f \in \mathbf{B}_{n,k}$ such that $G = \mathbf{S}(f)$. For each $b < k$ define as follows a 2-valued function $f_b: 2^n \to \{b, k\}$:

$$f_b(x) = \begin{cases} b & \text{if } f(x) = b \\ k & \text{if } f(x) \neq b. \end{cases}$$

It is straightforward to show that

$$\mathbf{S}(f) = \mathbf{S}(f_0) \cap \cdots \cap \mathbf{S}(f_{k-1}).$$

But also, conversely, we can prove that (2)⇒(1). Indeed, assume that $f_b \in \mathbf{B}_n$, $b < k$, is a given family of boolean valued functions such that G is the intersection of the strongly representable groups $\mathbf{S}(f_b)$. Define $f \in \mathbf{B}_{n,2^k}$ as follows:

$$f(x) = \langle f_0(x), \cdots, f_{k-1}(x) \rangle,$$

where for any integers $n_0, \cdots, n_{k-1}$, the symbol $\langle n_0, \cdots, n_{k-1} \rangle$ represents a standard coding of the k-tuple $(n_0, \cdots, n_{k-1})$. It is then clear that $\mathbf{S}(f) = \mathbf{S}(f_0) \cap \cdots \cap \mathbf{S}(f_{k-1})$, as desired.

To prove that (3) is equivalent to statements (1) and (2) it is enough to show that (i) for any family $\{f_i: 0 \leqq i \leqq k\}$ of boolean functions $f_i \in \mathbf{B}_n$ there exists an integer $0 \leqq m \leqq \log k$ and a boolean function $f \in \mathbf{B}_{n+m}$ such that

$$(\mathbf{S}(f))_{\{n+1, \cdots, n+m\}} = \mathbf{S}(f_1) \cap \cdots \cap \mathbf{S}(f_k), \tag{4}$$

and (ii) also conversely, for any integer $m \geqq 0$, and any boolean function $f \in \mathbf{B}_{n+m}$ there exist boolean functions $\{f_i: 0 \leqq i \leqq k\}$, with $k \leqq 2^m$ such that equation (1) holds.

Indeed, part (i) of the above statement follows by repeated application of part (6) of Theorem 10 and the case $k = 2$ of the above statement. To prove the case $k = 2$, define $f(x_1, \cdots, x_n, i) = f_i(x_1, \cdots, x_n)$. The desired equality is now easily proved. To prove the converse part (ii), let m, f be as in the hypothesis and define the desired family of functions $f_{b_1, \cdots, b_m}$ as follows.

$$f_{b_1, \cdots, b_m}(x_1, \cdots, x_n) = f(x_1, \cdots, x_n, b_1, \cdots, b_m).$$

It is now easy to see that equation (1) is satisfied. This completes the proof of the theorem. □

5.2. Representation theorems for general permutation groups. Here we study the representability problem for general permutation groups, give a necessary and sufficient condition via Pólya's cycle index for a permutation group to be representable, and show that the notions of representable and strongly representable coincide. In order to state the first general representation theorem we define, for any $n + 1 \leqq \theta \leqq 2^n$ and any permutation group $G \leqq \mathbf{S}_n$, the set $\mathbf{G}_\theta^{(n)} = \{M \leqq G: \Theta_n(M) = \theta\}$. Also, for any $H \subseteq \mathbf{S}_n$, and any $g \in \mathbf{S}_n$, the notation $\langle H, g \rangle$ denotes the least subgroup of $\mathbf{S}_n$ containing the set $H \cup \{g\}$.

THEOREM 13 (Representation Theorem). *The following statements are equivalent for any permutation groups $H < G \leqq \mathbf{S}_n$.*

(1) *$H = G \cap K$, for some strongly representable permutation group $K \leqq \mathbf{S}_n$.*
(2) *$H = G \cap K$, for some representable permutation group $K \leqq \mathbf{S}_n$.*
(3) *(for all $g \in G - H$)$(\Theta_n(\langle H, g \rangle) < \Theta_n(H))$.*
(4) *H is maximal in $\mathbf{G}_\theta^{(n)}$, where $\Theta_n(H) = \theta$.*

Proof. We prove the equivalence of the above statements by showing the following sequence of implications: (1)⇒(2)⇒(3)⇒(1) and (4)⇒(3)⇒(4). The proof of (1)⇒ (2) is trivial. First, we prove (2)⇒(3). By Theorem 12, K is the intersection of a family

of strongly representable groups. Hence, by assumption let $\mathbf{S}(f_i)$, where $\{f_i\} \subseteq \mathbf{B}_n$, be a finite family of invariance groups such that

$$H = \bigcap_i \mathbf{S}(f_i) \cap G.$$

Assume on the contrary that there exists an $H\ K \leqq G$ such that $\Theta(K) = \Theta(H)$. This last statement is equivalent to the statement

$$\forall x \in 2^n \ (x^K = x^H).$$

We show that in fact

$$K \subseteq \bigcap_i \mathbf{S}(f_i) \cap G,$$

which is a contradiction, since the right-hand side of the above inequality is equal to H. Indeed, let $\sigma \in K$ and $x \in 2^n$. Then we know that

$$x^K = (x^\sigma)^K = (x^\sigma)^H.$$

It follows that $x = (x^\sigma)^\tau$, for some $\tau \in H$. Consequently, $f_i(x) = f_i((x^\sigma)^\tau) = f_i(x^\sigma)$, as desired.

Next we prove that (3)$\Rightarrow$(1). Let $P_n(X)$ be the property of subgroups stated by $X \leqq \mathbf{S}_n \wedge (\text{for all } L > X)(\Theta_n(L) < \Theta_n(X))$. (When n and $X \leqq S_n$ are clear from context, we say simply that X satisfies property P.)

CLAIM. *For all n and subgroups X of* $\mathbf{S}_n$,

$$P_n(X) \Leftrightarrow X \text{ is strongly representable.}$$

Proof. As the direction from right to left is obvious, we only consider the direction from left to right. Suppose, in order to obtain a contradiction, that this direction fails. Let $X \leqq \mathbf{S}_n$ be of *maximal* size such that $P_n(X)$ holds, but that X is not strongly representable. It follows that

$$(\forall L > X)(L \text{ satisfies } P \Rightarrow L \text{ is strongly representable}).$$

Since the full symmetric group $\mathbf{S}_n$ is strongly representable we can assume, without loss of generality, that $X < \mathbf{S}_n$. In particular, there is a strongly representable group $L > X$ of *minimal* size. Let $h \in \mathbf{B}_n$ be such that $L = \mathbf{S}(h)$. Thus,

$$(*) \qquad \forall M(X < M < L \Rightarrow M \text{ does not satisfy } P).$$

Since $P_n(X)$ holds, we have that $\Theta_n(L) < \Theta_n(K)$. It follows that there exist $x, y \in 2^n$ such that

$$x = y \bmod L, \qquad x \neq y \bmod X,$$

where for $H \leqq \mathbf{S}_n$ and $x, y \in 2^n$ the symbol $x = y \bmod H$ means that $y = x^\sigma$, for some $\sigma \in H$. Define a boolean function $g \in \mathbf{B}_n$ as follows, for $w \in 2^n$,

$$g(w) = \begin{cases} h(w) & \text{if } w \neq x \bmod X, \quad w \neq y \bmod X \\ 0 & \text{if } w = x \bmod X \\ 1 & \text{if } w = y \bmod X. \end{cases}$$

It follows from the definition of g that $X \leqq \mathbf{S}(g) < \mathbf{S}(h) = L$. Since every strongly representable group satisfies property P, an immediate consequence of $(*)$ is that $X = \mathbf{S}(g)$. This completes the proof of the claim. $\square$

Now returning to the proof of (3)⇒(1), by assumption, for all $g \in G-H$, $2^{\Theta_n(\langle H,g\rangle)} < 2^{\Theta_n(H)}$. In particular, for all $g \in G-H$, there exists a boolean function $f_g \in \mathbf{B}_n$ such that $H \leqq \mathbf{S}_n(f_g)$, but $\langle H, g\rangle$ is not a subset of $\mathbf{S}_n(f_g)$. Consider the representable group K defined by

$$K = \bigcap_{g \in G-H} \mathbf{S}(f_g).$$

It is now trivial to check that $H = K \cap G$. Moreover, as in the implication (2)⇒(3) above, it follows that the permutation group K satisfies property P. By the above claim, K is strongly representable. This concludes the proof (3)⇒(1).

It remains to prove the equivalence of the last statement of the theorem. First we prove (4)⇒(3). Assume that H is a maximal element of $\mathbf{G}_\theta^{(n)}$, but that for some $g \in G-H$, we have that $\Theta_n(\langle H, g\rangle) = \Theta_n(H)$. But then $H < \langle H, g\rangle \leqq G$, contradicting the maximality of H. Finally, we prove (3)⇒(4). Assume on the contrary that (3) is true but that H is not maximal in $\mathbf{G}_\theta^{(n)}$. This means there exists $H < K \leqq G$ such that $\Theta_n(K) = \Theta_n(H)$. Take any $g \in K-H$ and notice that

$$\Theta_n(\langle H, g\rangle) \geqq \Theta_n(K) = \theta = \Theta_n(H) \geqq \Theta_n(\langle H, g\rangle).$$

Hence, $\Theta_n(H) = \Theta_n(\langle H, g\rangle)$, contradicting (3). □

A "naive" algorithm for testing the representability of a general permutation group $G \leqq \mathbf{S}_n$ is to test all boolean functions $f \in \mathbf{B}_n$ to see if $G = \mathbf{S}_n(f)$. Clearly, this requires time 2^{2^n}. An immediate consequence of the representation theorem is the following algorithm whose running time is $O((n!)^2) = 2^{O(n \log n)}$.

Algorithm for Deciding the Representability of Permutation Groups Input
A permutation group $G \leqq \mathbf{S}_n$.
for each $\sigma \in \mathbf{S}_n - G$ **do**
  if $\Theta_n(\langle G, \sigma\rangle) = \Theta_n(G)$
  then output G is not representable.
od
else output G is representable.
end

The well-known graph nonisomorphism problem (**NGIP**) is related to the above group representation problem. Indeed, let

$$G = (\{v_1, \cdots, v_n\}, E_G), \qquad H = (\{u_1, \cdots, u_n\}, E_H)$$

be two graphs on n vertices each. Consider the permutation group $ISO(G, H) \leqq \mathbf{S}_{n+3}$ whose generators σ satisfy:

$$\forall 1 \leqq i, \qquad j \leqq n (E_G(v_i, v_j) \Leftrightarrow E_H(u_{\sigma(i)}, u_{\sigma(j)})),$$

and in addition the permutation $n+i \to \sigma(n+i)$, $i = 1, 2, 3$, belongs to the group $C_3 = (n+1, n+2, n+3)$. It is easy to show that if G, H are isomorphic, then there exists a group $K \leqq \mathbf{S}_n$ such that $ISO(G, H) = K \times C_3$. On the other hand, if G, H are not isomorphic, then $ISO(G, H) = \langle id_{n+3}\rangle$. As a consequence of the nonrepresentability of C_3, and the representability theorem of direct products, it follows that G, H are not isomorphic if and only if $ISO(G, H) = \langle id_{n+3}\rangle$.

Remark. An idea similar to that used in the proof of the representation theorem can also be used to show that for any representable permutation groups $G < H \leqq \mathbf{S}_n$,

$$2 \cdot |\{h \in \mathbf{B}_n : H = \mathbf{S}(h)\}| \leqq |\{g \in \mathbf{B}_n : G = \mathbf{S}(g)\}|.$$

Indeed, assume that G, H are as above. Without loss of generality we may assume that there is no representable group K such that $G < K < H$. As in the proof of the representation theorem there exist $x, y \in 2^n$ such that $x = y \bmod H$, $x \neq y \bmod G$. Define two boolean functions $h_b \in \mathbf{B}_n$, $b = 0, 1$, as follows from $w \in 2^n$,

$$h_b(w) = \begin{cases} h(w) & \text{if } w \neq x \bmod G, \quad w \neq y \bmod G \\ b & \text{if } w = x \bmod G \\ \bar{b} & \text{if } w = y \bmod G. \end{cases}$$

Since $G \leqq \mathbf{S}(h_b) < \mathbf{S}(h)$, it follows from the above definition that each $h \in \mathbf{B}_n$ with $H = \mathbf{S}(h)$ gives rise to two distinct $h_b \in \mathbf{B}_n$, $b = 0, 1$, such that $G = \mathbf{S}(h_b)$. Moreover, it is not difficult to check that the mapping $h \to \{h_0, h_1\}$, where $H = \mathbf{S}(h)$, is $1-1$. It is now easy to complete the proof of the assertion.

An immediate consequence of the representation theorem is that all cycle indices $\Theta_n(G)$ can in fact be realized by representable permutation groups. The previous theorem also has a consequence concerning the representation of "maximal" permutation groups.

THEOREM 14 (Maximality Theorem). (1) *If H is a maximal proper subgroup of $G \leqq \mathbf{S}_n$ then*

$$\Theta_n(G) < \Theta_n(H) \Leftrightarrow (\exists f \in \mathbf{B}_n)(H = G \cap \mathbf{S}(f)).$$

(2) *All maximal subgroups of $\mathbf{S}_n$ are strongly representable, the only exceptions being*: (a) *the alternating group $\mathbf{A}_n$, for all $n \geqq 3$*; (b) *the 1-dimensional, linear, affine group $AGL_1(5)$ over the field of five elements, for $n = 5$*; (c) *the group of linear transformations $PGL_2(5)$ of the projective line over the field of five elements, for $n = 6$*; (d) *the group of semilinear transformations $P\Gamma L_2(8)$ of the projective line of the field of eight elements, for $n = 9$.*

Proof. To prove (1) let H be a maximal proper subgroup of G such that $\Theta_n(G) < \Theta_n(H)$. Put $\theta = \Theta_n(H)$. Since condition (4) of the representation theorem is satisfied, H is of the form $\mathbf{S}(f)$, for some $f \in \mathbf{B}_n$. This completes the proof of ($\Rightarrow$). To prove the other direction, assume that $\Theta_n(G) = \Theta_n(H)$. Then for all $g \in G - H$, $\Theta_n(\langle H, g \rangle) = \Theta_n(H)$. Hence, again by the representation theorem, there is no $f \in \mathbf{B}_n$ such that $H = G \cap \mathbf{S}(f)$. This completes the proof of (1).

To prove (2) let M be a maximal subgroup of $\mathbf{S}_n$. We distinguish two cases.

Case 1. $\Theta_n(M) > n+1$.

In this case, part (1) of this theorem implies that M is strongly representable, since $\Theta_n(\mathbf{S}_n) = n+1$. (Note that by Theorem 2(4), the condition of Case 1 is satisfied by all intransitive groups M, i.e., groups with $\omega_n(M) \geqq 2$.)

Case 2. $\Theta_n(M) = n+1$.

In this case we know from the main theorem of [BP55] that M is of one of the forms in the statement of the theorem. □

As noted above, all maximal permutation groups with the exception of $\mathbf{A}_n$ are of the form $\mathbf{S}(f)$, provided that $n \geqq 10$. Such maximal permutation groups include: the cartesian products $\mathbf{S}_k \times \mathbf{S}_{n-k}$ ($k \leqq n/2$), the wreath products $\mathbf{S}_k \wr \mathbf{S}_l$ ($n = kl$, $k, l > 1$), the affine groups $AGL_d(p)$, for $n = p^d$, etc. The interested reader will find a complete survey of classification results for maximal permutation groups in [KL88]. It should also be pointed out that there are plenty of nonmaximal permutation groups which are not representable. In fact, it can be verified that examples of such groups are the wreath products $G \wr \mathbf{A}_n$. In general we can prove the following theorem. For any permutation groups $G \leqq \mathbf{S}_m$, $H \leqq \mathbf{S}_n$.

THEOREM 15. *Let $G \leqq \mathbf{S}_m$, $H \leqq \mathbf{S}_n$. Then*

(1) *G and H representable $\Rightarrow G \wr H$ is representable.*

(2) *$G \wr H$ is representable $\Rightarrow H$ is representable.*

(3) *$G \wr H$ is representable and $2^n < m \Rightarrow G$ is weakly representable.*

(4) *For p prime, a p-Sylow subgroup P of $\mathbf{S}_n$ is representable $\Leftrightarrow p \neq 3, 4, 5$.*

Proof. (1) Suppose we are given two representable groups $G = \mathbf{S}(L_G) \leqq \mathbf{S}_m$, $H = \mathbf{S}(L_H) \leqq \mathbf{S}_n$, where $L_G \subseteq \{0, 1\}^m$, $L_H \subseteq \{0, 1\}^n$. We want to show that the wreath product $G \wr H \leqq \mathbf{S}_{mn}$ is representable. The wreath product $G \wr H$ consists of all permutations $\rho = [\sigma; \tau_1, \cdots, \tau_m]$, where $\sigma \in G$ and $\tau_1, \cdots, \tau_m \in H$, such that

$$\rho((k-1)n + i) = \sigma(k)n + \tau_{\sigma(k)}(i),$$

for $1 \leqq k \leqq m$, $1 \leqq i \leqq n$. (Intuitively speaking, ρ acts on $m \times n$ matrices in such a way that τ_i acts only on the ith row and σ permutes rows.) Without loss of generality we can assume that $0^m, 1^m \in L_G$ and $0^n, 1^n \in L_H$. Define a set $L \subseteq \{0, 1\}^{mn}$ of words w by the disjunctoin of the following three clauses:

(a) $|w|_1 = n$, and for some $0 \leqq k < m$, $w_{kn+1} = \cdots = w_{kn+n} = 1$ (i.e., the $(k+1)$st row consists only of 1's).

(b) $|w|_1 > n$, and w is of the form $e_1^n e_2^n \cdots e_m^n$, where the word $e_1 e_2 \cdots e_m \in L_G$.

(c) $|w|_1 > n$ and w is not of the form $e_1^n e_2^n \cdots e_m^n$, but $w_{kn+1} \cdots w_{kn+n} \in L_H$, for all $0 \leqq k < m$.

We claim that $\mathbf{S}_{mn}(L) = G \wr H$. Indeed, the inequality $G \wr H \subseteq \mathbf{S}_{mn}(L)$ is clear. To prove the other direction assume that $\rho \in \mathbf{S}_{mn}(L)$. By clause (a), ρ respects the n-blocks of words of length mn. Hence, ρ is of the form $\rho = [\sigma; \tau_1, \cdots, \tau_m]$, and $\tau_i \in \mathbf{S}_n$, $\sigma \in G$, where $i = 1, \cdots, m$. If $\sigma \notin G$, then there is a word v of length m, with $v \in L_G$ and $v^\sigma \notin L_G$. Then (using clause (b) above) we have that $w = v_1^n v_2^n \cdots v_m^n \in L$, but $w^\rho \notin L$, which is a contradiction. If for some i, $\tau_i \notin H$, then there is a word v of length n such that $v \in L_H$ and $v^{\tau_i} \notin L_H$. It follows (by clause (c) above) that the word $w = v \cdots v \in L$, but $w^\rho \notin L$, a contradiction. This completes the proof of (1).

(2) By assumption, $G \wr H = \mathbf{S}_{mn}(f)$, for some $f \in \mathbf{B}_{mn}$. Hence,

$$G \wr H = \{[\sigma; \tau_1, \cdots, \tau_m] \in \mathbf{S}_m \wr \mathbf{S}_n\colon (\forall X_1, \cdots, X_m) f(X_{\sigma(1)}^{\tau_1}, \cdots, X_{\sigma(m)}^{\tau_m}) = f(X_1, \cdots, X_m)\}.$$

In particular, we have that

$$\begin{aligned} \tau \in H &\Leftrightarrow [id_m; \tau, id_n, \cdots, id_n] \in G \wr H \\ &\Leftrightarrow \forall X_1 [\forall X_2, \cdots, X_m (f_{X_2, \cdots, X_m}(X_1^\tau) = f_{X_2, \cdots, X_m}(X_1))] \\ &\Leftrightarrow \tau \in \bigcap_{X_2, \cdots, X_m \in 2^n} \mathbf{S}(f_{X_2, \cdots, X_m}), \end{aligned}$$

as desired.

The proof of (3) is similar and uses the simple observation that for any permutation $\sigma \in \mathbf{S}_m$,

$$[\sigma; id_n, \cdots, id_n] \in G \wr 1 \Leftrightarrow (\forall X_1, \cdots, X_m) f(X_{\sigma(1)}, \cdots, X_{\sigma(m)}) = f(X_1, \cdots, X_m).$$

(4) Let p be a prime $p \leqq n$. By Sylow's theorem, all the p-Sylow subgroups of $\mathbf{S}_n$ are conjugates of one another. Moreover, by [Pas66, pp. 8–11], if C is the cyclic group $(1, 2, \cdots, p)$, then there exists an integer r such if we iterate the wreath product r times on C then the group $C \wr C \cdots \wr C$ obtained is a p-Sylow subgroup of $\mathbf{S}_n$. Combining this with the previous assertions of the theorem, as well as part (3) of Theorem 10, we obtain the desired result. □

The converse of part (1) of the above theorem is not necessarily true. This is easy to see from the following example. We show that the wreath product $\mathbf{A}_3 \wr \mathbf{S}_2$ is representable, but that $\mathbf{A}_3$ is not. Indeed, consider the language

$$L = \{001101, 010011, 110100, 001110, 100011, 111000\} \subseteq 2^6.$$

We already proved that $\mathbf{A}_3$ is not representable. We claim that $\mathbf{A}_3 \wr \mathbf{S}_2 = \mathbf{S}_6(L)$. Consider the three-cycle $\tau = (\{1, 2\}, \{3, 4\}, \{5, 6\})$. It is easy to see $\mathbf{A}_3 \wr \mathbf{S}_2$ consists of the 24 permutations σ in $\mathbf{S}_6$ which permute the two-element sets $\{1, 2\}, \{3, 4\}, \{5, 6\}$ as in the three-cycles τ, τ^2, τ^3. A straightforward (but tedious) computation shows that $\mathbf{S}_6(L)$ also consists of exactly the above 24 permutations.

Another class of examples of nonrepresentable groups is given by the direct products of the form $\mathbf{A}_m \times G$, $G \times \mathbf{A}_m$, where G is any permutation group acting on a set which is disjoint from $\{1, 2, \cdots, m\}$, $m \geqq 3$ (for a proof of this, see the next subsection).

We conclude this section by showing the representability of the normalizers of groups G generated by a family of "disjoint" transpositions. Let G be a subgroup of $\mathbf{S}_n$ and let $H = \langle H(x) \colon x \in 2^n \rangle$ be a family of normal subgroups of $N(G)$ (the normalizer of G in $\mathbf{S}_n$) such that for all $\sigma \in N(G)$, $x \in 2^n$, $H(x) = H(\sigma(x))$. (This last condition is satisfied if, for example, each $H(x) = 1$ or each $H(x) = G$.) For any $x \in 2^n$ let $G_x = \{\sigma \in G \colon x^\sigma = x\}$ be the stabilizer of G at x. Define the function $f_{G,H} \colon 2^n \to 2$ as follows:

$$f_{G,H}(x) = \begin{cases} 1 & \textit{if } G_x = H(x) \\ 0 & \textit{if } G_x \neq H(x). \end{cases}$$

Normalizers of certain permutation groups can be written in the form $\mathbf{S}(f)$. To see this observe the following two claims.

(1) $N(G) \subseteq \mathbf{S}(f_{G,H})$.

(2) If (for all $\sigma \in \mathbf{S}_n$) [(for all $x \in 2^n$) $(G_x = H(x) \Leftrightarrow G_{\sigma(x)} = H(x)) \Rightarrow G^\sigma = G$] then there exists an $f \in \mathbf{B}_n$ such that $N(G) = \mathbf{S}(f)$.

For convenience, let $\sigma(x)$ denote x^σ. To prove (1) let $\sigma \in N(G)$. This means that $G^\sigma = G$. We want to show that

$$\forall x \in 2^n (G_x = H(x) \Leftrightarrow G_{\sigma(x)} = H(x)).$$

To prove the implication ($\Rightarrow$) notice that

$$H(x) = G_x = (G^\sigma)_x = (G_{\sigma(x)})^\sigma = H(x)^\sigma.$$

Hence, $H(x) = G_{\sigma(x)}$, as desired. The converse ($\Leftarrow$) is similar.

The proof of assertion (2) is immediate. The hypothesis is simply a restatement of the condition $\mathbf{S}(f_{G,H}) \subseteq N(G)$.

5.3. A logspace algorithm for the representability of cyclic groups. This section is devoted to the proof of the existence and correctness of a logspace algorithm which, when given as input a cyclic group $G \leqq \mathbf{S}_n$, decides whether the group is representable, in which case it outputs a boolean function $f \in \mathbf{B}_{n,k}$ such that $G = \mathbf{S}(f)$. The algorithm is as follows.

Algorithm for Representing Cyclic Groups

Input

$G = \langle \sigma \rangle$ cyclic group.

Step 1

Decompose $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k$, where $\sigma_1, \sigma_2, \cdots, \sigma_k$ are disjoint cycles of lengths $l_1, l_2, \cdots, l_k \geqq 2$, respectively.

Step 2
if for all $1 \leqq i \leqq k$,
$l_i = 3 \Rightarrow (\exists j \neq i)(3 \mid l_j)$ **and**
$l_i = 4 \Rightarrow (\exists j \neq i)(\gcd(4, l_j) \neq 1)$ **and**
$l_i = 5 \Rightarrow (\exists j \neq i)(5 \mid l_j)$
then output G is representable.
else output G is not representable.
end

At the present time, we do not know how to efficiently test the representability of arbitrary abelian groups (or other natural classes of groups such as solvable, nilpotent, etc.). *If* a given abelian group K can be decomposed into *disjoint* cyclic factors, *then* we have the following NC algorithm for testing representability: (1) use an NC algorithm [LM85], [MC85], [Mul86] to "factor" K into its cyclic factors and then (2) apply the "cyclic-group" algorithm to each of the cyclic factors of K. In view of the lemma below, the group K is representable exactly when each of its disjoint, cyclic factors is.

LEMMA 16. *Let $G \leqq \mathbf{S}_m$, $H \leqq \mathbf{S}_n$ be permutation groups. Then $G \times H$ is representable $\Leftrightarrow$ both G, H are representable.*

Proof. ($\Rightarrow$) By the representability of the groups G, H there exist boolean functions $f \in \mathbf{B}_m$ and $g \in \mathbf{B}_n$ such that $G \times H = \mathbf{S}(f) \times \mathbf{S}(g)$. By the maximality theorem there exists a function $h: 2^{m+n} \to 2$ such that $\mathbf{S}(h) = \mathbf{S}_m \times \mathbf{S}_n$. Hence, if we put $F(x, y) = \langle f(x), g(y) \rangle$, then it is easy to see that

$$\mathbf{S}(f) \times \mathbf{S}(g) = \mathbf{S}(h) \cap \mathbf{S}(F).$$

This implies that $G \times H$ is representable, and hence also strongly representable.

To prove ($\Leftarrow$) assume that $G \times H = \mathbf{S}(f)$, for some $f: 2^{m+n} \to k$. It is then easy to see that

$$\begin{aligned} G &= \{\sigma \in \mathbf{S}_m : \langle \sigma, id_n \rangle \in G \times H\} \\ &= \{\sigma \in \mathbf{S}_m : (\forall x, y)(f(x^\sigma, y) = f(x, y))\} \\ &= \{\sigma \in \mathbf{S}_m : (\forall y)(f_y^\sigma = f_y)\} \\ &= \bigcap_{y \in 2^n} \mathbf{S}(f_y). \end{aligned}$$

A similar proof works for the group H. □

The main result of the present section is the following theorem.

THEOREM 17 (Cyclic Group Representability Theorem). *There is a logspace algorithm which, when given as input a cyclic group $G \leqq \mathbf{S}_n$, decides whether the group is representable, in which case it outputs a function $f \in \mathbf{B}_n$ such that $G = \mathbf{S}(f)$.*

The rest of this section is dedicated to the proof (sketch) of correctness of the above algorithm. The proof is in a series of lemmas. For technical reasons, we introduce two definitions. A boolean function $f \in \mathbf{B}_n$ is called *special* if for all words w of length n,

$$|w|_1 = 1 \Rightarrow f(w) = 1.$$

Let $\sigma_1, \cdots, \sigma_k$ be a collection of cycles. We say that the group $G = \sigma_1, \cdots, \sigma_k$, generated by the permutations $\sigma_1, \cdots, \sigma_k$, is *specially representable* if there exists a special boolean function $f: 2^\Omega \to 2$ (where Ω is the union of the supports of the σ_i's) such that $G = \mathbf{S}(f)$. The support of a permutation σ, denoted by $Supp(\sigma)$, is the set

of i such that $\sigma(i) \neq i$. The support of a permutation group G, denoted $Supp(G)$, is the union of the supports of the elements of G.

5.4. Main ideas of the proof. Before proceeding with the details, it will be instructive to give an outline of the main ideas needed for the corectness proof. We are given a cyclic group G generated by a permutation σ. Decompose σ into disjoint cycles $\sigma_1, \sigma_2, \cdots, \sigma_k$ of lengths $l_1, l_2, \cdots, l_k \geqq 2$, respectively.

If $k=1$ then we know that G is specially representable exactly when $l_1 \neq 3, 4, 5$. (The representability of the cyclic group C_s, for $s \neq 3, 4, 5$ is proved in § 4; for $s=3, 4, 5$ observe that for any $f \in \mathbf{B}_s$, if $C_s \subseteq \mathbf{S}(f)$ then $D_s \subseteq \mathbf{S}(f)$, where D_s is the dihedral group. We refrain from repeating the proof and refer the reader to § 4 for the details.)

If $k=2$ then the result will follow by considering several possibilities for the pairs (l_1, l_2):

- if $\gcd(l_1, l_2)=1$, then $G=\langle\sigma_1\rangle \times \langle\sigma_2\rangle$ is the direct product of σ_1 and σ_2. Hence, G is specially representable exactly when both factors are specially representable,
- if $(l_1, l_2)=(3,3)$ or $(4,4)$ or $(5,5)$ then G is specially representable,
- if $(l_1, l_2)=(3, m)$ (with $3|m$) or $(4, m)$ (with $\gcd(4, m) \neq 1$) or $(5, m)$ (with $5|m$), then G is specially representable.

This will take care of deciding the representability of G for all possible pairs (l_1, l_2). A similar argument will work for $k \geqq 3$. This concludes the outline of the proof of correctness.

5.4.1. Sketch of proof. The details of the above constructions are rather tedious but a sufficient indication is given in the sequel.

LEMMA 18. *Suppose that $\sigma_1, \cdots, \sigma_{n+1}$ is a collection of cycles such that both $\langle\sigma_1, \cdots, \sigma_n\rangle$ and $\langle\sigma_{n+1}\rangle$ are specially representable and have disjoint supports. Then $\langle\sigma_1, \cdots, \sigma_{n+1}\rangle$ is specially representable.*

Proof.

Put

$$\Omega_0 = \bigcup_{i=1}^{n} Supp(\sigma_i), \qquad \Omega_1 = Supp(\sigma_{n+1})$$

and let $|\Omega_0| = m$, $|\Omega_1| = k$. Suppose that $f_0: 2^{\Omega_0} \to 2$ and $f_1: 2^{\Omega_1} \to 2$ are special boolean functions representing the groups $\langle\sigma_1, \cdots, \sigma_n\rangle$ and $\langle\sigma_{n+1}\rangle$, respectively. Without loss of generality, we may assume that $1 = f_0(0^m) \neq f_1(0^k) = 0$. Let $\Omega = \Omega_0 \cup \Omega_1$ and define the function f: $2^\Omega \to 2$ by

$$f(w) = f_0(w \restriction \Omega_0) f_1(w \restriction \Omega_1).$$

Clearly, $\langle\sigma_1, \cdots, \sigma_{n+1}\rangle \subseteq \mathbf{S}_\Omega(f)$. Hence, it remains to prove that

$$\mathbf{S}_\Omega(f) \subseteq \langle\sigma_1, \cdots, \sigma_{n+1}\rangle.$$

Assume on the contrary that $\tau \in \mathbf{S}_\Omega(f) - \langle\sigma_1, \cdots, \sigma_{n+1}\rangle$. We distinguish two cases.

Case 1. $(\exists i \in \Omega_0)(\exists j \in \Omega_1)(\tau(i) = j)$.

Let $w \in \{0, 1\}^\Omega$ be defined by $w \restriction \Omega_0 = 0^m$, and

$$(w \restriction \Omega_1)(l) = \begin{cases} 0 & \text{if } l \neq j \\ 1 & \text{if } l = j, \end{cases}$$

for $l \in \Omega_1$. Since f is a special boolean function and using the fact that $f_0(0^m) \neq f_1(0^k)$ we obtain that $f(w) = 1 \neq f(w^\tau) = 0$, which is a contradiction.

Case 2. (For all $i \in \Omega_0)(\tau(i) \in \Omega_0)$.

Put $\tau_0 = (\tau \upharpoonright \Omega_0) \in \mathbf{S}_{\Omega_0}$ and $\tau_1 = (\tau \upharpoonright \Omega_1) \in \mathbf{S}_{\Omega_1}$. By hypothesis, for all $w \in 2^{\Omega}$, we have that

$$f(w) = f_0(w \upharpoonright \Omega_0) f_1(w \upharpoonright \Omega_1) = f(w^\tau) = f_0((w \upharpoonright \Omega_0)^{\tau_0}) f_1((w \upharpoonright \Omega_1)^{\tau_1}),$$

which implies $\tau_0 \in \mathbf{S}^+_{\Omega_0}(f_0)$ and $\tau_1 \in \mathbf{S}^+_{\Omega_1}(f_1)$. This completes the proof of the lemma.

An immediate consequence of the previous lemma is the following.

LEMMA 19. *If G, H have disjoint support and are specially representable then $G \times H$ is specially representable.*

Next we will be concerned with the problem of representing cyclic groups. In view of Theorem 7 in § 4, we know that the cyclic group $\langle(1, 2, \cdots, n)\rangle$ is representable exactly when $n \neq 3, 4, 5$. In particular, the groups $\langle(1, 2, 3)\rangle$, $\langle(1, 2, 3, 4)\rangle$, $\langle(1, 2, 3, 4, 5)\rangle$ are not representable. The following lemma may be somewhat surprising, since it implies that the group $\langle(1, 2, 3)(4, 5, 6)\rangle$, though isomorphic to $\langle(1, 2, 3)\rangle$, *is* representable.

LEMMA 20. *Let the cyclic group G be generated by a permutation σ which is the product of two disjoint cycles of lengths l_1, l_2, respectively. Then G is specially representable exactly when the following conditions are satisfied*: $(l_1 = 3 \Rightarrow 3 \mid l_2)$ *and* $(l_2 = 3 \Rightarrow 3 \mid l_1)$, $(l_1 = 4 \Rightarrow \gcd(4, l_2) \neq 1)$ *and* $(l_2 = 4 \Rightarrow \gcd(4, l_1) \neq 1)$, $(l_1 = 5 \Rightarrow 5 \mid l_2)$ *and* $(l_2 = 5 \Rightarrow 5 \mid l_1)$.

Sketch of proof. It is clear that the assertion of the lemma will follow if we can prove that the three assertions below are true.

(1) The groups $\langle(1, 2, \cdots, n)(n+1, n+2, \cdots, kn)\rangle$ are specially representable when $n = 3, 4, 5$.

(2) The groups $\langle(1, 2, 3, 4)(5, \cdots, m+4)\rangle$ are specially representable when $\gcd(4, m) \neq 1$.

(3) Let m, n be given integers such that either $m = n = 2$ or $m = 2$ and $n \geqq 6$ or $n = 2$ and $m \geqq 6$ or $m, n \geqq 6$. Then $\langle(1, 2, \cdots, m)(m+1, m+2, \cdots, m+n)\rangle$ is specially representable.

Proof of (1). We give the proof only for the case $n = 5$ and $k = 2$. The other cases $n = 3$, $n = 4$, and $k \geqq 3$ are treated similarly. Details of these constructions are left to the reader. Let $\sigma = \sigma_0 \sigma_1$, where $\sigma_0 = (1, 2, 3, 4, 5)$ and $\sigma_1 = (6, 7, 8, 9, 10)$. From the proof of Theorem 7 in § 4 we know that

$$D_5 = \mathbf{S}_5(L') = \mathbf{S}_5(L''),$$

where $L' = 0^*1^*0^* \cup 1^*0^*1^*$ and $L'' = \{w \in L' : |w|_0 \geqq 1\}$. Let L consist of all words w of length 10 such that

—either $|w|_1 = 1$

—or $|w|_1 = 2$ and $(\exists 1 \leqq i \leqq 5)$ $(w_i = w_{5+i}$ and $(\forall j \neq i, 5+i)$ $(w_j = 0))$

—or $|w|_1 = 3$ and $(\exists 0 \leqq i \leqq 4)$ $(w = (1000011000)^{\sigma^i}$ or $w = (1100010000)^{\sigma^i})$

—or $|w|_1$ 3 and $w_1 \cdots w_5 \in L'$ and $w_6 \cdots w_{10} \in L''$.

We want to show that in fact $\langle(1, 2, 3, 4, 5)(6, 7, 8, 9, 10)\rangle = \mathbf{S}_{10}(L)$. It is clear that

$$\langle(1, 2, 3, 4, 5)(6, 7, 8, 9, 10)\rangle \subseteq \mathbf{S}_{10}(L).$$

Conversely, suppose that $\tau \in \mathbf{S}_{10}(L)$. Assume on the contrary there exists an $1 \leqq i \leqq 5$ and a $6 \leqq j \leqq 10$ such that $\tau(i) = j$. Let the word w be defined such that $w_l = 0$, if $l = j$, and $= 1$ otherwise. It follows from the last clause in the definition of L and the fact that $0^5 \notin L''$ that $w \notin L$ and $w^\tau \in L$, contradicting the assumption $\tau \in \mathbf{S}_{10}(L)$. It follows that τ is the product of two disjoint permutations τ_0 and τ_1 acting on $1, 2, \cdots, 5$ and $6, 7, \cdots, 10$, respectively. It follows from the last clause in the definition of L that $\tau_0 \in D_5$ and $\tau_1 \in \pi^{-1} D_5 \pi$, where $\pi(i) = 5 + i$, for $i = 1, \cdots, 5$. Let $\rho_0 = (1, 5)(2, 4)$ and

$\rho_1=(6,10)(7,9)$ be the reflection permutations on $1,2,\cdots,5$ and $6,7,\cdots,10$, respectively. To complete the proof of (1), it is enough to show that none of the permutations

$$\rho_0,\ \rho_1,\ \rho_0\rho_1,\ \rho_0\sigma_1^i,\ \sigma_0^i\rho_1,\ \sigma_0^i\sigma_1^j,$$

for $i\neq j$, belong to $\mathbf{S}_{10}(L)$. To see this, let $x=1000011000\in L$. Then for the permutations $\tau=\rho_0,\rho_1,\rho_0\rho_1,\rho_0\sigma_1^i$ for $i=1,2,3,5$, and $\tau=\sigma_0^i\rho_1$ for $i=1,2,4,5$ it is easy to check that $x^\tau\notin L$. Let $x=110001000$. Then for $\tau=\rho_0\sigma_1^4$ and $\tau=\sigma_0^3\rho_1$ it is easy to check that $x^\tau\notin L$. Finally, for $x=1000010000\in L$ and $\sigma_0^i\sigma_1^j$, where $i\neq j$, we have that $x^\tau\notin L$. This completes the proof of part (1) of the lemma.

Proof of (2). Put $\sigma_0=(1,2,3,4)$, $\sigma_1=(5,6,\cdots,m+4)$, $\sigma=\sigma_0\sigma_1$. Let L be the set of words of length $m+4$ such that

either $|w|_1=1$
or $|w|_1=2$ and $(\exists 0\leqq i\leqq \mathrm{lcm}\,(4,m)-1)(w=(100010^{m-1})^{\sigma^i})$
or $|w|_1=3$ and $(\exists 0\leqq i\leqq \mathrm{lcm}\,(4,m)-1)(w=(110010^{m-1})^{\sigma^i})$
or $|w|_1=3$ and $w_1\cdots w_4\in L'$ and $w_5\cdots w_{m+5}\in L''$,

where $L'=0^*1^*0^*\cup 1^*0^*1^*$ and L'' are as in Theorem 7 of § 4 satisfying $\mathbf{S}_m(L'')=C_m$ and moreover for all $i\geqq 1$, $0^i\notin L''$. Clearly, $\langle(1,2,3,4)(5,6,\cdots,m+4)\rangle\subseteq\mathbf{S}_{m+4}(L)$. It remains to prove that

$$\mathbf{S}_{m+4}(L)\subseteq\langle(1,2,3,4)(5,6,\cdots,m+4)\rangle.$$

Let $\tau\in\langle(1,2,3,4)(5,6,\cdots,m+4)\rangle$. As before, $\tau=\tau_0\tau_1$, where $\tau_0\in D_4$ and $\tau_1\in\pi^{-1}D_m\pi$, where $\pi(i)=4+i$ for $i=1,2,\cdots,m$. Let $\rho=(1,4)(2,3)$ be the reflection on $1,2,3,4$. It suffices to show that none of the permutations

$$\rho\sigma_1^i,\ \sigma_0^i\sigma_1^j,$$

for $i\neq j \bmod 4$ are in $\mathbf{S}_{m+4}(L)$. Indeed, if $\tau=\sigma_0^i\sigma_1^j$, then let $x=100010^{m-1}$. So it is clear that $x\in L$, but $x^\tau\notin L$. Next assume that $\tau=\rho\sigma_1^i$. We distinguish the following two cases.

Case 1. $m=4k$, i.e., a multiple of 4.

Let $x=100010^{m-1}$. Then $x\in L$, but $x^\tau\notin L$ unless $x^\tau=x^{\sigma^j}$ for some j. In this case $j=3 \bmod 4$ and $j=i \bmod 4k$. So it follows that $i=3,7,11,\cdots,4k-1$. Now let $y=110010^{m-1}$. Then $y\in L$, but $y^\tau\notin L$ for the above values of i, unless $y^\tau=y^{\sigma^l}$ for some l. In that case we have that $l=2 \bmod 4$ *and* $l=i \bmod 4k$. So it follows that $i=2,6,10,\cdots,4k-2$. Consequently, $\tau\notin\mathbf{S}_{m+4}(L)$.

Case 2. $\gcd\,(4,m)=2$.

Let $x=100010^{m-1}$. Then $x\in L$, but $x^\tau\notin L$ unless $x^\tau=x^{\sigma^j}$ for some j. In this case $j=3 \bmod 4$ and $j=i \bmod 4k$. So it follows that for even values of i, $\tau\notin\mathbf{S}_{m+4}(L)$. Let $y=110010^{m-1}$. Then $y\in L$, but $y^\tau\notin L$ unless $y^\tau=y^{\sigma^l}$ for some l. In that case we have that $l=2 \bmod 4$ and $l=i \bmod m$. So it follows that for odd values of i, $\tau\notin\mathbf{S}_{m+4}(L)$. This completes the proof of (2).

Proof of (3). A similar technique can be used to generalize the representability result to more general types of cycles. Details are left as an exercise to the reader.

A straightforward generalization of Lemma 20 is given in the next lemma.

LEMMA 21. *Let G be a permutation group generated by a permutation σ which can be decomposed into k-many disjoint cycles of lengths $l_1,l_2,\cdots,l_k$, respectively. The group G is specially representable exactly when the following conditions are satisfied for all* $1\leqq i\leqq k$,

$$l_i=3\Rightarrow(\exists j\neq i)(3\,|\,l_j)\quad\textbf{and}$$
$$l_i=4\Rightarrow(\exists j\neq i)(\gcd\,(4,l_j)\neq 1)\quad\textbf{and}$$
$$l_i=5\Rightarrow(\exists j\neq i)(5\,|\,l_j).$$

Now the correctness of the algorithm is an immediate consequence of Lemmas 1–5. This completes the proof of Theorem 17. □

5.5. Asymptotic behavior. Finally, for any sequence $\langle G_n \leqq \mathbf{S}_n \colon n \geqq 1\rangle$ of permutation groups we consider the value of the limit

$$\lim_{n\to\infty} \frac{|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) = G_n\}|}{2^{2^n}}.$$

We have the following theorem.

THEOREM 22. (Almost all boolean functions have trivial invariance groups.) *For any family $\langle G_n \colon n \geqq 1\rangle$ of permutation groups such that each $G_n \leqq \mathbf{S}_n$, we have that*

$$\lim_{n\to\infty} \frac{|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) = \{id_n\}\}|}{2^{2^n}} = \lim_{n\to\infty} \frac{|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) \leqq G_n\}|}{2^{2^n}} = 1.$$

Moreover, if $\liminf |G_n| > 1$ *then*

$$\lim_{n\to\infty} \frac{|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) \geqq G_n\}|}{2^{2^n}} = \lim_{n\to\infty} \frac{|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) = G_n\}|}{2^{2^n}} = 0.$$

Proof. During the course of this proof we use the abbreviation $\Theta(m) := \Theta_m(\langle(1, 2, \cdots, m)\rangle)$. First we prove the second part of the theorem. By assumption there exists an n_0 such that for all $n \geqq n_0$, $|G_n| > 1$. Hence, for each $n \geqq n_0$, G_n contains a permutation of order $k(n) \geqq 2$, say σ_n. Without loss of generality we can assume that each $k(n)$ is a prime number. Since $k(n)$ is prime, σ_n is a product of $k(n)$-cycles. If $(i_1, \cdots, i_{k(n)})$ is the first $k(n)$-cycle in this product then it is easy to see that

$$\Theta_n(\langle\sigma_n\rangle) \leqq \Theta_n(\langle(i_1, \cdots, i_{k(n)})\rangle).$$

It follows that

$$|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) \geqq G_n\}| \leqq |\{f \in \mathbf{B}_n \colon \sigma_n \in \mathbf{S}(f)\}| = 2^{\Theta_n(\sigma_n)} \leqq 2^{\Theta(k(n)) \cdot 2^{n-k(n)}}.$$

$$|\{f \in \mathbf{B}_n \colon \mathbf{S}(f) \geqq G_n\}| \leqq |\{f \in \mathbf{B}_n \colon \sigma_n \in \mathbf{S}(f)\}| = 2^{\Theta_n(\sigma_n)} \leqq 2^{\Theta(k(n)) \cdot 2^{n-k(n)}}.$$

Recall from [Ber71] that the formula

$$\Theta(m) = \frac{1}{m} \cdot \sum_{k|m} \phi(k) \cdot 2^{m/k}$$

gives the Pólya cycle index of the group $\langle(1, 2, \cdots, m)\rangle$ acting on $\{1, 2, \cdots, m\}$, where $\phi(k)$ is Euler's totient function. However, it is easy to see that for k prime

$$\frac{\Theta(k)}{2^k} = \frac{1}{k} + \frac{2}{2^k} - \frac{2}{k2^k}.$$

In fact the function in the right-hand side of the above equation is decreasing in k. Hence, for k prime,

$$\frac{\Theta(k)}{2^k} \leqq \frac{\Theta(2)}{2^2} = \frac{3}{4}.$$

It follows that

$$\frac{|\{f\in \mathbf{B}_n: \mathbf{S}(f)\geqq G_n\}|}{2^{2^n}}\leqq 2^{2^n\cdot[\Theta(k(n))\cdot 2^{-k(n)}-1]}\leqq 2^{-2^{n-2}}.$$

Since the right-hand side of the above inequality converges to 0 the proof of the second part of the theorem is complete. To prove the first part notice that

$$\{f\in \mathbf{B}_n: \mathbf{S}(f)\neq id_n\}\subseteq \bigcup_{\sigma\neq id_n}\{f\in \mathbf{B}_n: \sigma\in \mathbf{S}(f)\},$$

where σ ranges over cyclic permutations of order a prime number $\leqq n$. Since there are at most $n!$ permutations on n letters we obtain from the last inequality that

$$\frac{|\{f\in \mathbf{B}_n: \mathbf{S}(f)\neq\{id_n\}\}|}{2^{2^n}}\leqq n!\cdot 2^{-2^{n-2}}=2^{O(n\log n)}\cdot 2^{-2^{n-2}}\to 0,$$

as desired. □

As a consequence of the above theorem we obtain that asymptotically almost all boolean functions have trivial invariance group.

6. Invariance groups of languages and circuits. In this section we classify languages according to the size of their invariance groups. Furthermore, we consider questions concerning their structural properties and complexity. Recall that for each $L\subseteq\{0,1\}^*$ and n, L_n is the set of strings in L of length exactly n. By abuse of notation we also denote the characteristic function of L_n with the same symbol. Let $\mathbf{S}_n(L)$ denote the invariance group of the n-ary boolean function L_n. For any language L and any sequence $\sigma=\langle\sigma_n: n\geqq 1\rangle$ of permutations such that each $\sigma_n\in\mathbf{S}_n$ we define the language

$$L_n^\sigma=\{x\in 2^n: x^{\sigma_n}\in L_n\}.$$

For each n let $G_n\leqq\mathbf{S}_n$ and put $\mathbf{G}=\langle G_n: n\geqq 1\rangle$. Define

$$L^{\mathbf{G}}=\bigcup_{\sigma_n\in G_n} L_n^{\sigma_n}.$$

For each $1\leqq k\leqq\infty$, let $\mathbf{F}_k$ be the class of functions $n^{c\log^{(k)}n}$, $c>0$, where $\log^{(1)} n=\log n$, $\log^{(k+1)} n=\log\log^{(k)} n$, and $\log^{(\infty)} n=1$. Clearly, $\mathbf{F}_\infty$ is the class $\mathbf{P}$ of polynomial functions. We also define $\mathbf{F}_0$ as the class of functions 2^{cn}, $c>0$. Let $\mathbf{L}(\mathbf{F}_k)$ be the set languages $L\subseteq\{0,1\}^*$ such that there exists a function $f\in\mathbf{F}_k$ satisfying

$$\forall n(|\mathbf{S}_n: \mathbf{S}_n(L)|\leqq f(n)).$$

We will also use the notation $L(\mathbf{EXP})$ and $L(\mathbf{P})$ for the classes $\mathbf{L}(\mathbf{F}_0)$ and $\mathbf{L}(\mathbf{F}_\infty)$, respectively. Occasionally, a language $L\in L(\mathbf{P})$ will also be called a language which has *polynomial index* or is even *almost symmetric.*

6.1. Structural properties. The following theorem gives some of the structural properties of the classes of languages $\mathbf{L}(\mathbf{F}_k)$.

THEOREM 23. *For any $0\leqq k\leqq\infty$ and any language $L\in\mathbf{L}(\mathbf{F}_k)$,*

(1) *$\mathbf{L}(\mathbf{F}_k)$ is closed under boolean operations and homomorphisms,*

(2) *$(L\cdot\Sigma)\in\mathbf{L}(\mathbf{F}_k)$,*

(3) *$L^\sigma\in\mathbf{L}(\mathbf{F}_k)$, where $\sigma=\langle\sigma_n: n\geqq 1\rangle$, with each $\sigma_n\in\mathbf{S}_n$,*

(4) *if $|\mathbf{S}_n: N_{\mathbf{S}_n}(G_n)|\leqq f(n)$ and $f\in\mathbf{F}_k$ then $L^{\mathbf{G}}\in\mathbf{L}(\mathbf{F}_k)$, where $\mathbf{G}=\langle G_n: n\geqq 1\rangle$.*

Proof. We use extensively (even without explicit mention) the results of Theorem 10. To prove (1) notice first that $\mathbf{S}_n(\neg L)=\mathbf{S}_n(L)$. To prove that $\mathbf{L}(\mathbf{F}_k)$ is closed under union and intersection use the following inequality from group theory: for $K, K'\leqq G$,

$$|G:K\cap K'|\leqq|G:K|\cdot|G:K'|.$$

For example, for closure under intersection we have that $\mathbf{S}_n(L)\cap\mathbf{S}_n(L')\subseteq\mathbf{S}_n(L\cap L')$, which implies that

$$|\mathbf{S}_n:\mathbf{S}_n(L\cap L')|\leqq|\mathbf{S}_n:\mathbf{S}_n(L)\cap\mathbf{S}_n(L')|\leqq|\mathbf{S}_n:\mathbf{S}_n(L)|\cdot|\mathbf{S}_n:\mathbf{S}_n(L')|.$$

To prove closure under a homomorphism $h:L\to L'$ note that $\mathbf{S}_n(L)\subseteq\mathbf{S}_n(h(L))$. Hence,

$$|\mathbf{S}_n:\mathbf{S}_n(L')|=|\mathbf{S}_n:\mathbf{S}_n(h(L))|\leqq|\mathbf{S}_n:\mathbf{S}_n(L)|.$$

To prove (2) let $L'=L\cdot\Sigma=\{xa: x\in L,\ a\in\Sigma\}$ and note that

$$|\mathbf{S}_n:\mathbf{S}_n(L')|\leqq n\cdot|\mathbf{S}_{n-1}:\mathbf{S}_{n-1}(L)|.$$

To prove (3) note that $\mathbf{S}_n(L)^{\sigma_n}=\mathbf{S}_n(L^{\sigma})$. To prove (4), note that we have $N_{\mathbf{S}_n}(G_n)\cap\mathbf{S}_n(L)\subseteq\mathbf{S}_n(L^{\mathbf{G}})$. Indeed, for $\tau\in N_{\mathbf{S}_n}(G_n)\cap\mathbf{S}_n(L)$ we have that $G_n\tau=\tau G_n$, which in turn implies that

$$L_n^{G_n\tau}=L_n^{\tau G_n}=\bigcup_{\sigma_n\in G_n}L_n^{\tau\sigma_n}=\bigcup_{\sigma_n\in G_n}L_n^{\sigma_n}=L_n^{G_n}.$$

Hence,

$$|\mathbf{S}_n:\mathbf{S}_n(L^{\mathbf{G}})|\leqq|\mathbf{S}_n:N(G_n)|\cdot|\mathbf{S}_n:\mathbf{S}_n(L)|,$$

as desired. □

The classes $L(\mathbf{P})$ and $L(\mathbf{EXP})$ enjoy the closure properties mentioned below.

THEOREM 24.

$$L\in L(\mathbf{P})\quad\textit{and}\quad p\in\mathbf{P}\Rightarrow|\mathbf{S}_{p(n)}:\mathbf{S}_{p(n)}(L)|=n^{O(1)}.$$

Proof. The proof is obvious, since the class of polynomials is closed under composition. □

THEOREM 25.

$$L^1, L^2\in L(\mathbf{EXP})\Rightarrow L=\{xy: x\in L^1, y\in L^2, l(x)=l(y)\}\in L(\mathbf{EXP}).$$

Proof. It is clear that $\mathbf{S}_n(L^1)\times\mathbf{S}_n(L^2)\subseteq\mathbf{S}_{2n}(L)$. It follows from Stirling's formula that

$$\begin{aligned}|\mathbf{S}_{2n}:\mathbf{S}_{2n}(L)|&\leqq\frac{(2n)!}{|\mathbf{S}_n(L)|\cdot|\mathbf{S}_n(L)|}\\&=\frac{(2n)!}{n!\cdot n!}\cdot|\mathbf{S}_n:\mathbf{S}_n(L)|^2\\&\leqq\frac{(2n)!}{n!\cdot n!}\cdot 2^{O(n)}=2^{O(n)}.\end{aligned}$$ □

Let **REG** denote the class of regular languages.

THEOREM 26. *The following properties hold for any* $1\leqq k<\infty$,

$$\mathbf{L}(\mathbf{F}_\infty)=L(\mathbf{P})\subset\cdots\subset\mathbf{L}(\mathbf{F}_{k+1})\subset\mathbf{L}(\mathbf{F}_k)\subset\cdots\subset L(\mathbf{EXP})=\mathbf{L}(\mathbf{F}_0),\tag{1}$$

$$\mathbf{REG}\cap L(\mathbf{P})\neq\varnothing,\quad \mathbf{REG}-L(\mathbf{EXP})\neq\varnothing,\quad L(\mathbf{P})-\mathbf{REG}\neq\varnothing.\tag{2}$$

Proof. To prove $\mathbf{L}(\mathbf{F}_{k+1})\subset\mathbf{L}(\mathbf{F}_k)$, for $1\leqq k<\infty$, put $f(n)=n-\log^{(k)}n$ and consider the language

$$L=\{x\in 2^n: x_{f(n)+1}\leqq\cdots\leqq x_n\}.$$

Then we have that

$$|\mathbf{S}_n:\mathbf{S}_n(L)|=\frac{n!}{f(n)!}=n^{O(\log^{(k)} n)}.$$

It follows that $\mathbf{L}(\mathbf{F}_{k+1})\subset\mathbf{L}(\mathbf{F}_k)$. (Note that by the pumping lemma for regular languages L cannot be regular.) The proof of $\mathbf{L}(\mathbf{F}_k)\subset\mathbf{L}(\mathbf{F}_0)$ is more delicate. The group $\mathbf{S}_n\times\mathbf{S}_n$ is maximal in $\mathbf{S}_{2n}$. It follows from our representation theorem for maximal groups that there exists a language L such that for all n,

$$\mathbf{S}_{2n}(L)=\mathbf{S}_n\times\mathbf{S}_n.$$

It follows from Stirling's formula that $|\mathbf{S}_{2n}:\mathbf{S}_{2n}(L)|=2^{O(n)}$, as desired. The proof of $\mathbf{L}(\mathbf{F}_\infty)\subset\mathbf{L}(\mathbf{F}_k)$, $k\geqq 1$, follows from the above remarks. This completes the proof of (1). To prove $\mathbf{REG}\cap L(\mathbf{P})\neq\emptyset$, consider the trivial language $L=\{0,1\}^*$. To prove $\mathbf{REG}-L(\mathbf{EXP})\neq\emptyset$, consider the language $L=0^*1^*$. To prove $L(\mathbf{P})-\mathbf{REG}\neq\emptyset$. For any set S of positive integers let $L^S=\{0^n\colon n\in S\}$. Clearly, $L_n^S(x)=1$ if $n\in S$ and $x=0^n$, and $=0$ otherwise. It is easy to see that for all S, $L^S\in L(\mathbf{P})$, and hence $L(\mathbf{P})$ is uncoutable. (In fact, $\mathbf{S}_n(L^S)=\mathbf{S}_n$, for all n and S.) In particular, the nonregular language $L=\{0^p\colon p$ is a prime number$\}\in L(\mathbf{P})$. □

A few useful and illuminating examples are now in order.

Examples. (1) Let $L^k=\{x\in\{0,1\}^*\colon l(x)\geqq k,\ x_1\leqq\cdots\leqq x_k\}$. Then $\mathbf{S}_n(L^k)=\mathbf{S}_{n-k}$ and therefore $|\mathbf{S}_n:\mathbf{S}_n(L)|=n!/(n-k)!=O(n^k)$. Hence, for all k, $L^k\in L(\mathbf{P})$.

(2) For each word $x=x_1\cdots x_n$ let $x^T=x_n\cdots x_1$ and $L^T=\{x^T\colon x\in L\}$. Put $\sigma_n(i)=n-i+1$. Then $L^\sigma=L^T$, where $\sigma=\langle\sigma_n\colon n\geqq 1\rangle$.

(3) There exist languages $L^0, L^1\in L(\mathbf{P})$ such that $L^0\cdot L^1\notin L(\mathbf{EXP})$. Indeed, put $L^0=\{0\}^*$, $L^1=\{1\}^*$. Then $L=L^0\cdot L^1=\{0^n1^m\colon n,m\geqq 0\}$. It is easy to see that $|\mathbf{S}:\mathbf{S}_n(L)|=n!$.

(4) There exists a language $L\in L(\mathbf{P})$ such that $L^*\notin L(\mathbf{P})$. Indeed, put $L=\{01\}$. Then for n even, $\sigma\in\mathbf{S}$ if and only if for all $i\leqq n$ (i is even if and only if $\sigma(i)$ is even). It follows that $|\mathbf{S}_n:\mathbf{S}_n(L)|=n!/(n/2)!(n/2)!$. Hence, $L^*\in L(\mathbf{EXP})-L(\mathbf{P})$.

(5) $L(\mathbf{P})$ is not closed under inverse homomorphism. Indeed, let D be the Dyck language on one parenthesis and $h\colon D\to L$ be the homomorphism $h(0)=h(1)=0$. In view of the results of § 3, $D\notin L(\mathbf{P})$.

(6) For each function $f\colon\mathbf{N}\to\mathbf{N}$ such that for all $n\geqq 1$, $f(n)\leqq n$, we define the language

$$L_n^f=\{x\in 2^n\colon x_1\leqq\cdots\leqq x_{f(n)}\},\qquad L^f=\bigcup_n L_n^f.$$

Using the pumping lemma for regular languages we can show that $L^f\in\mathbf{REG}\Rightarrow\sup_n f(n)<\infty$.

Similar classes of languages corresponding to the cycle index can be defined as follows. Let $L_\Theta(\mathbf{F}_k)$ be the set of languages L such that there exists a function $f\in\mathbf{F}_k$ satisfying

$$\forall n(\Theta(\mathbf{S}(L_n))\leqq f(n)).$$

Since, $\Theta(\mathbf{S}_n(L))\leqq(n+1)\cdot|\mathbf{S}_n:\mathbf{S}_n(L)|$, it is clear that $\mathbf{L}(\mathbf{F}_k)\subseteq L_\Theta(\mathbf{F}_k)$. In fact we can show that $\mathbf{L}(\mathbf{F}_k)\subset L_\Theta(\mathbf{F}_k)$. To see this take $f(n)=n-\log^{(k)} n$. Define $x\in L_n$ if and only if $x_1\leqq x_2\leqq\cdots\leqq x_{f(n)}$. Then it is easy to see that $\mathbf{S}_n(L)=\mathbf{S}_{f(n)}$. Hence, $|\mathbf{S}_n:\mathbf{S}_n(L)|=O(n^{\log n})$, while $\Theta(\mathbf{S}_n(L))=(f(n)+1)2^{\log^{(k)} n}=O(n^2)$.

6.2. Circuit complexity of formal languages. In this section, we study the complexity of languages $L\in L(\mathbf{P})$. The following result is proved by applying the intricate

NC algorithm of [BLS87] for permutation group membership. By delving into a deep result in classification theory of finite simple groups, we improve the conclusion to that of Theorem 29. For clarity however, we present the following.

THEOREM 27. *For any language* $L \subseteq \{0,1\}^*$, *if* $L \in L(\mathbf{P})$ *then* L *is nonuniform* NC.

Proof. As a first step in the proof we will need the following claim.

CLAIM. *There is an* NC^1 *algorithm which, when given* $x \in \{0,1\}^n$, *outputs* $\sigma \in \mathbf{S}_n$ *such that* $x^\sigma = 1^m 0^{n-m}$, *for some* m.

Proof of the claim. Before giving the proof of the claim, we illustrate the idea by citing an example. Suppose that $x = 101100111$. By simultaneously going from left to right and from right to left, we swap an "out-of-place" 0 with an "out-of-place" 1, keeping track of the respective positions.[3] This gives rise to the desired permutation σ. In the case at hand we find $\sigma = (2,9)(5,8)(6,7)$ and $x^\sigma = 1^6 0^3$.

Now we proceed with the proof of the main claim. Define the predicates $E_{k,b}(u)$, to hold when there are exactly k occurrences of b in the word u ($b = 0, 1$) are in NC^1. The predicates $E_{k,b}$ are obviously computable in constant depth, polynomial size threshold circuits, i.e., in TC^0. By the work of Ajtai, Komlós, and Szemerédi [AKS83] $\mathrm{TC}^0 \subseteq \mathrm{NC}^1$. For $k = 1, \cdots, [n/2]$ and $1 \leqq i < j \leqq n$, let $\alpha_{i,j,k}$ be a log depth circuit which outputs 1 exactly when the kth "out-of-place" 0 is in position i and the kth "out-of-place" 1 is in position j. It follows that $\alpha_{i,j,k}(x) = 1$ if and only if "there exist $k-1$ zeros to the left of position i, the ith bit of x is zero, and there exist k ones to the right of position i" and "there exist $k-1$ ones to the right of position j, the jth bit of x is one, and there exist k zeros to the left of position j." This in turn is equivalent to

$$E_{k-1,0}(x_1, \cdots, x_{i-1}) \quad \text{and } x_i = 0 \quad \text{and } E_{k,1}(x_{i+1}, \cdots, x_n) \quad \text{and}$$
$$E_{k-1,1}(x_{j+1}, \cdots, x_n) \quad \text{and } x_j = 1 \quad \text{and } E_{k,0}(x_1 \cdots x_{j-1}).$$

This implies that the required permutation can be defined by

$$\sigma = \prod \left\{ (i,j) \colon i < j \text{ and } \bigvee_{k=1}^{[n/2]} \alpha_{i,j,k} \right\}.$$

Converting the fanin, $[n/2]$-$\vee$-gate into a $\log([n/2])$ depth tree of fanin, 2-$\vee$-gates, we have an NC^1 procedure for computing σ. This completes the proof of the claim.

Next we continue with the proof of the main theorem. Put $G_n = \mathbf{S}_n(L)$ and let $R_n = \{h_1, \cdots, h_q\}$ be a complete set of representatives for the left cosets of G_n, where $q \leqq p(n)$ and $p(n)$ is a polynomial such that $|\mathbf{S}_n : G_n| \leqq p(n)$. Fix $x \in \{0,1\}^n$. By the previous claim there is a permutation σ which is the product of disjoint transpositions and an integer $0 \leqq k \leqq n$ such that $x^\sigma = 1^k 0^{n-k}$. So $x = (1^k 0^{n-k})^\sigma$. In parallel for $i = 1, \cdots, q$ test whether $h_i^{-1}\sigma \in G_n$ by using the principal result of [BLS87], thus determining i such that $\sigma = h_i g$, for some $g \in G_n$. Then we obtain that

$$L_n(x) = L_n((1^k 0^{n-k})^\sigma) = L_n((1^k 0^{n-k})^{h_i g}) = L_n((1^k 0^{n-k})^{h_i}).$$

By hardwiring the polynomially many values $L_n((1^k 0^{n-k})^{h_i})$ for $0 \leqq k \leqq n$ and $1 \leqq i \leqq q$, we produce a polynomial size polylogarithmic depth circuit family for L. □

Theorem 27 involves a straightforward application of the beautiful NC algorithm of Babai, Luks, and Seress [BLS87] for testing membership in a finite permutation group. By using the deep structure consequences of the O'Nan–Scott theorem below, together with Bochert's result on the size of the index of primitive permutation groups

---

[3] This is a well-known trick for improving the efficiency of the "partition" or "split" algorithm used in quick-sort.

(see Theorem 1(3) in § 2), we can improve the NC algorithm of Theorem 27 to an optimal TC^0 algorithm (and hence NC^1). First, we take the following discussion and statement of the O'Nan-Scott theorem from [KL88, p. 376].

Let $I=\{1,2,\cdots,n\}$ and let $\mathbf{S}_n$ act naturally on I. Consider all subgroups of the following five classes of subgroups of $\mathbf{S}_n$.

α_1: $\mathbf{S}_k\times\mathbf{S}_{n-k}$, where $1\leqq k\leqq n/2$,

α_2: $\mathbf{S}_a \wr \mathbf{S}_b$, where either ($n=ab$ and $a, b \quad 1$) or ($n=a^b$ and $a\geqq 5$, $b\geqq 2$),

α_3: the affine groups $AGL_d(p)$, where $n=p^d$,

α_4: $T^k\cdot(Out(T)\times\mathbf{S}_k)$, where T is a nonabelian simple group, $k\geqq 2$ and $n=|T|^{k-1}$, as well as all groups in the class,

α_5: almost simple groups acting primitively on I.

THEOREM 28 (O'Nan-Scott). *Every subgroup of* $\mathbf{S}_n$ *not containing* $\mathbf{A}_n$ *is a member of* $\alpha_1\cup\cdots\cup\alpha_5$.

Now we can improve the result of Theorem 27 in the following way.

THEOREM 29 (Parallel complexity of Languages of Polynomial Index). *For any language* $L\subseteq\{0,1\}^*$, *if* $L\in L(\mathbf{P})$ *then* L *is in* 9*-nonuniform* TC^0 *and hence in (nonuniform)* NC^1.

Proof. The proof requires the following consequence of the O'Nan-Scott theorem.

CLAIM. *Suppose that* $\langle G_n\leqq\mathbf{S}_n\colon n\geqq 1\rangle$ *is a family of permutation groups such that for all* n, $|\mathbf{S}_n:G_n|\leqq n^k$, *for some* k. *Then for sufficiently large* N, *there exists an* $i_n\leqq k$ *for which* $G_n=U_n\times V_n$ *with the supports of* U_n, V_n *disjoint and* $U_n\leqq\mathbf{S}_{i_n}$, $V_n=\mathbf{S}_{n-i_n}$.

Before proving the claim we complete the details of the proof of Theorem 29. Apply the claim to $G_n=\mathbf{S}_n(L)$ and notice that given $x\in 2^n$, the question of whether x belongs to L is decided completely by the number of 1's in the support of $K_n=\mathbf{S}_{n-i_n}$, together with information about the action of a finite group $H_n\leqq\mathbf{S}_{i_n}$, for $i_n\leqq k$. Using the counting predicates as in the proof of Theorem 27, it is clear that this is a TC^0 and hence NC^1 algorithm. Thus, the proof of the theorem is complete, assuming the claim.

Proof of the claim. We have already observed at the beginning of § 5 that $G_n\neq\mathbf{A}_n$. By the O'Nan-Scott theorem, G_n is a member of $\alpha_1\cup\cdots\cup\alpha_5$. Using Bochert's theorem on the size of the index of primitive permutation groups (§ 2, Theorem 1(3)), the observations of [LPS88] concerning the primitivity of the maximal groups in $\alpha_3\cup\alpha_4\cup\alpha_5$ and the fact that G_n has polynomial index with respect to $\mathbf{S}_n$, we conclude that the subgroup G_n cannot be a member of the class $\alpha_3\cup\alpha_4\cup\alpha_5$. It follows that $G_n\in\alpha_1\cup\alpha_2$. We show that in fact $G_n\notin\alpha_2$. Assume on the contrary that $G_n\leqq H_n=\mathbf{S}_a\wr\mathbf{S}_b$. It follows that $|H_n|=a!(b!)^a$. We distinguish the following two cases.

Case 1. $n=ab$, for $a,b>1$.

In this case it is easy to verify using Stirling's interpolation formula

$$(n/e)^n\sqrt{n}<n!<(n/e)^n 3\sqrt{n}$$

that

$$|\mathbf{S}_n:H_n|=\frac{n!}{a!(b!)^a}\sim\frac{a^{n-a}}{3b^{a/2}(3/a)^a\sqrt{a}}.$$

Moreover, it is clear that the right-hand side of this last inequality cannot be asymptotically polynomial in n, since $a\leqq n$ is a proper divisor of n, which is a contradiction.

Case 2. $n=a^b$, for $a\geqq 5$, $b\geqq 2$.

A similar calculation shows that asymptotically

$$|\mathbf{S}_n:H_n|=\frac{n!}{a!(b!)^a}=\frac{n!}{a!(b'!)^a},$$

where $b' = a^{b-1}$. It follows from the argument of Case 1 that this last quantity cannot be asymptotically polynomial in n, which is a contradiction. It follows that $G_n \in \alpha_1$. Let $G_n \leqq \mathbf{S}_i \times \mathbf{S}_{n-i}$, for some $1 \leqq i_n n/2$. We claim that, in fact, $i_n \leqq k$ for all but a finite number of n's. Indeed, put $i_n = i$ and notice that

$$|\mathbf{S}_n : \mathbf{S}_i \times \mathbf{S}_{n-i}| = \frac{n!}{i!(n-i)!} = \Omega(n^i) \leqq |\mathbf{S}_n : G_n| \leqq n^k,$$

which proves that $i \leqq k$. It follows that $G_n = U_n \times V_n$, where $U_n \leqq \mathbf{S}_{i_n}$ and $V_n \leqq \mathbf{S}_{n-i_n}$. Since $i_n \leqq k$ and $|\mathbf{S}_n : G_n| \leqq n^k$ it follows that for n large enough $V_n = \mathbf{S}_{n-i_n}$. This completes the proof of the claim. Now let $L \subseteq \{0, 1\}^*$ have polynomial index. Given a word $x \in \{0, 1\}^n$, in TC^0 one can test whether the number of 1's occurring in the $n - i_n$ positions (where $V_n = \mathbf{S}_{n-i_n}$) is equal to a fixed value, hardwired into the nth circuit. This, together with a finite look-up table corresponding to the U_n part, furnishes a TC^0 algorithm for testing membership in L. □

6.3. Applications. An immediate consequence of our analysis is that if $\langle G_n \leqq \mathbf{S}_n : n \geqq 1 \rangle$ is a family of transitive permutation groups such that $|\mathbf{S}_n : G_n| = n^{O(1)}$ then $G_n = \mathbf{S}_n$, for all but a finite number of n's (this answers a conjecture of Perrin). It is also possible to give a more algebraic formulation of the main consequence of Theorem 29. For p_n a polynomial in the variables $x_1, \cdots, x_n$ and with coefficients from the two element field $\mathbf{Z}_2$, let

$$\mathbf{S}(p_n) = \{\sigma \in \mathbf{S}_n : \forall x_1, \cdots, x_n (p_n(x_1, \cdots, x_n) = p_n(x_{\sigma(1)}, \cdots, x_{\sigma(n)}) \bmod 2)\}.$$

A family $\langle p_n : n \geqq 1 \rangle$ of multivariate polynomials in $\mathbf{Z}_2[x_1, \cdots, x_n]$ is of polynomial index if $|\mathbf{S}_n : \mathbf{S}(p_n)| = n^{O(1)}$.

THEOREM 30. *If $\langle p_n : n \geqq 1 \rangle$ is family of multivariate polynomials (in $\mathbf{Z}_2[x_1, \cdots, x_n]$) of polynomial index then there is a family $\langle q_n : n \geqq 1 \rangle$ of multivariate polynomials (in $\mathbf{Z}_2[x_1, \cdots, x_n]$) of polynomial length such that $p_n = q_n$.*

Because of the limitations of families of groups of polynomial index proved in the claim above, we obtain a generalization of the principal results of [FKPS85]. Namely, for $L \subseteq \{0, 1\}^*$ let $\mu_L(n)$ be the least number of input bits which must be set to a constant in order for the resulting language $L_n = L \cap \{0, 1\}^n$ to be constant (see [FKPS85] for more details). Then we can prove the following theorem.

THEOREM 31. *If $L \in L(\mathbf{P})$* (i.e., *L is a language of polynomial index*) *then*

$$\mu_L(n) \leqq (\log n)^{O(1)} \Leftrightarrow L \in \mathrm{AC}^0.$$

Our characterization of permutation groups of polynomial index given during the proof of Theorem 29 can also be used to determine the parallel complexity of the following problem concerning "weight-swapping." Let $\mathbf{G} = \langle G_n : n \in \mathbf{N} \rangle$ denote a sequence of permutation groups such that $G_n \leqq \mathbf{S}_n$, for all n. By SWAP ($\mathbf{G}$) we understand the following problem:

Input. $n \in \mathbf{N}$, $a_1, \cdots, a_n$ positive rationals, each of whose (binary) representations is of length at most n.

Output. A permutation $\sigma \in G_n$ such that for all $1 \leqq i\ n$, $a_{\alpha(i)} + a_{\sigma(i+1)} \leqq 2$, if such a permutation exists, and the response "NO" otherwise.

THEOREM 32. *For any sequence $\mathbf{G}$ of permutation groups of polynomial index, the problem* SWAP ($\mathbf{G}$) *is in nonuniform* NC^1.

Proof. By the characterization of sequences of groups of polynomial index, there exist integers k, N such that for all $n \geqq N$, $G_n = H_n \times K_n$, where $H_n \leqq \mathbf{S}_{i_n}$ and $K_n = \mathbf{S}_{n-i_n}$,

with $i_n \leqq k$. Given $n \geqq N$, and n positive rational weights $a_1, \cdots, a_n$ test whether there exist permutations $\sigma \in H_n$ and $\tau \in K_n$ such that for $1 \leqq i \leqq n$, $a_{(\sigma \times \tau)(i)} + a_{(\sigma + \tau)(i+1)} \leqq 2$, as follows. For τ, sort the set of weights $\{a_i: i \in Supp(K_n)\}$ in decreasing order. Assume wlog that $Supp(K_n) = \{1, \cdots, n - i_n\}$. Let $\rho \in K_n$ be a "sorting" permutation such that $a_{\rho(1)} \geqq a_{\rho(2)} \geqq \cdots \geqq a_{\rho(n-i_n)}$. Test in parallel whether

$$a_{\rho(1)} + a_{\rho(n-i_n)} \leqq 2, \qquad a_{\rho(2)} + a_{\rho(n-i_n-1)} \leqq 2, \cdots, \text{etc.}$$

If so, then let τ be the appropriate permutation such that

$$1 \mapsto \rho(1), \quad 2 \mapsto \rho(n-i_n), \cdots, n - i_n - 1 \mapsto \rho\left(\frac{n-i_n}{2} - 1\right), \quad n - i_n \mapsto \rho\left(\frac{n-i_n}{2}\right),$$

if $n - i_n$ is even, and a variant of this, if $n - i_n$ is odd. Since sorting n many n-bit numbers is in NC^1, computing τ is in NC^1. Since $H_n \leqq \mathbf{S}_{i_n}$, where $i_n \leqq k$, there are only a finite number of possibilities to test for σ. These are hardwired (by nonuniformity) into the circuit. □

The following conjecture would relate the cycle index of a sequence $\mathbf{G} = \langle G_n: n \geqq 1 \rangle$ of groups with the circuit complexity of the language L.

CONJECTURE 33. *For any language* $L \subseteq \{0, 1\}^*$, *if* $L \in L_\Theta(\mathbf{P})$ *then* L *is nonuniform* NC.

This conjecture appears somewhat plausible, since it follows from the next theorem that if $\mathbf{G} = \langle G_n \leqq \mathbf{S}_n: n \geqq 1 \rangle$ is a sequence of groups whose cycle index $\Theta_n(G_n)$, as a function of n, majorizes all polynomials, then there is a language L with $\mathbf{S}_n(L) \supseteq G_n$ and $L \notin SIZE(n^{O(1)})$.

THEOREM 34. *For any sequence* $\mathbf{G} = \langle G_n: n \geqq 1 \rangle$ *of permutation groups* $G_n \leqq \mathbf{S}_n$ *it is possible to find a language* L *such that*

$$L \notin SIZE(\sqrt{\Theta(G_n)}), \quad \text{and} \quad \forall n(\mathbf{S}(L_n) \supseteq G_n).$$

Proof. By Lupanov's theorem $|\{f \in \mathbf{B}_n: c(f) \leqq q\}| = O(q^{q+1}) = 2^{O(q \log q)}$. Hence, if $q_n \to \infty$ then $|\{f \in \mathbf{B}_n: c(f) \leqq q_n\}| < 2^{q_n^2}$. In particular, setting $q_n = \sqrt{\Theta(G_n)}$ we obtain

$$|\{f \in \mathbf{B}_n: c(f) \leqq \sqrt{\Theta(G_n)}\}| < 2^{\Theta(G_n)} = |\{f \in \mathbf{B}_n: \mathbf{S}(f) \supseteq G_n\}|.$$

It follows that for n big enough there exists an $f_n \in \mathbf{B}_n$ such that $\mathbf{S}(f_n) \supseteq G_n$ and $c(f_n) > \sqrt{\Theta(G_n)}$. This completes the proof of the theorem. □

7. Discussion and open problems. Three of the main questions we have tried to answer in the present paper are (1) which permutation groups arise as (or are isomorphic to) the invariance groups of boolean functions, (2) determining the complexity of deciding the representability of a permutation group, (3) determining the relation between the family of invariance groups of a formal language L and the parallel complexity of L.

Concerning question (1), we saw that most (i.e., with a few exceptions) maximal permutation subgroups of $\mathbf{S}_n$ are representable. We have shown that every permutation group $G\,\mathbf{S}_n$ is isomorphic to the invariance group of a boolean function $f \in \mathbf{B}_{n(\log n+1)}$. However, we do not know if this last "upper bound" can be improved to $f \in \mathbf{B}_{cn}$, for some constant c independent of n. In the case of question (2), we gave a logspace algorithm for deciding the representability of cyclic groups. In general however, we do not know of any efficient algorithm for deciding the representability of any other natural classes of permutation groups (e.g., abelian, nilpotent, solvable, etc.). The existence of a polynomial time algorithm for testing representability of an arbitrary permutation group is related to the question of whether *graph nonisomorphism* is in polynomial time.

Concerning question (3), we have shown a relation between the size of the index of the invariance group of a formal language and its complexity. We showed that any language of "polynomial size index" is in (nonuniform) TC^0. It is possible that a finer analysis of the structure results for maximal permutation groups will yield a similar result for other classes of languages, like the ones with subexponential or even exponential size index. We conjecture that a similar result is true for any language of "polynomial size Pólya index." We believe as well that there should be a relation between the algebraic structure of the syntactic monoid of a regular language $L \subseteq \{0, 1\}^*$ (Krohn–Rhodes theorem) and the family of invariance groups of L_n. As indicated by our preliminary work, straightforward approaches to such an investigation are not likely—the property of a group being representable is not preserved under homomorphism. Our parallel complexity results concern nonuniform families of boolean circuits. A natural sequel to our work might investigate uniform versions of some of our results. For instance, if $L \subseteq \{0, 1\}^*$ is a regular (or context free, or logspace computable, etc.) language with polynomial index (or polynomial size Pólya index) then is L in logspace uniform TC^0?

Another interesting question concerns the problem of giving an efficient algorithm A which on input a formal language L, a permutation $\sigma \in \mathbf{S}_n$, and an integer n, determines whether or not $\sigma \in \mathbf{S}_n(L)$, i.e.,

$$A(L, n, \sigma) = \begin{cases} 1 & \text{if } \sigma \in \mathbf{S}_n(L) \\ 0 & \text{otherwise.} \end{cases}$$

We investigated this question in the present paper for regular languages. The obvious algorithm has complexity $O(2^n)$ (to check membership of a permutation σ in $\mathbf{S}_n(L)$ test whether for all $x \in 2^n$, $x \in L_n \Leftrightarrow x^\sigma \in L_n$). A similar question applies to right-quotient representatives of $\mathbf{S}_n(L)$. It would also be interesting to investigate these questions for other types of languages, such as *CFL*, etc.

Acknowledgments. Discussions with Peter van Emde Boas, Danny Krizanc, Dominique Perrin, Paul Schupp, and Paul Vitányi are gratefully acknowledged. A. M. Cohen was extremely helpful with the literature on maximal permutation groups. Lambert Meertens made comments that significantly improved the presentation and pointed out that essentially our original n^2-upper-bound proof of the isomorphism theorem could yield the improved, $n(\log n + 1)$-upper-bound.

REFERENCES

[AKS83] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log(n))$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 1–9.

[Arb69] M. A. ARBIB, *Theories of Abstract Automata*, Prentice-Hall Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ, 1969.

[ASW85] C. ATTIYA, M. SNIR, AND M. WARMUTH, *Computing on an anonymous ring*, in Proc. 4th Annual ACM Symposium on Principles of Distributed Computation, 1985.

[BLS87] L. BABAI, E. LUKS, AND A. SERESS, *Permutation Groups in* NC, in Proc. 19th ACM Symposium on Theory of Computing, New York, 1987.

[BB89] P. BEAME AND H. BODLAENDER, *Distributed computing on transitive networks: The torus*, in Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS), 1989.

[BP55] R. A. BEAUMONT AND R. P. PETERSON, *Set-transitive permutation groups*, Canad. J. Math., 7 (1955), pp. 35–42.

[Ber71] C. BERGE, *Principles of Combinatorics*, Academic Press, New York, 1971.

[Com70] L. COMTET, *Analyse Combinatoire*, Deuxième Tome; Collection SUP, Presses Universitaires de France, 1970.

[Coo85] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[FKPS85] R. FAGIN, M. KLAWE, N. PIPPENGER, AND L. STOCKMEYER, *Bounded-Depth, Polynomial-Size Circuits for Symmetric Functions*, Theoret. Comput. Sci., 36 (1985), pp. 239-250.

[FKL88] L. FINKELSTEIN, D. KLEITMAN, AND T. LEIGHTON, *Applying the classification theorem for finite simple groups to minimize pin count in uniform permutation architectures*, VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing (AWOC)88, Corfu, Greece, June/July 1988, J. H. Reif, ed., Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, pp. 247-256.

[FHL80] M. FURST, J. HOPCROFT, AND E. LUKS, *Polynomial-time algorithms for permutation groups*, in Proc. 21st IEEE Symposium on Foundations of Computer Science, Syracuse, NY, 1980.

[FSS84] M. FURST, J. SAXE, AND M. SIPSER, *Parity circuits and the polynomial time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13-27.

[Hal57] M. HALL, *The Theory of Groups*, Macmillan, New York, 1957.

[Har64] M. HARRISON, *On the classification of Boolean functions by the general linear and affine groups*, J. Soc. Indust. Appl. Math., 12 (1964), pp. 285-299.

[Harr78] M. HARRISON, *Introduction to Formal Language Theory*, Addison Wesley, New York, 1978.

[KL82] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Tech. J., 49 (1982).

[KL88] P. B. KLEIDMAN AND M. W. LIEBECK, *A survey of the maximal subgroups of the finite simple groups*, in Geometries and Groups, M. Aschbacher, A. M. Cohen, and W. M. Kantor, eds., reprinted from Geometriae Dedicata 25(1-3), pp. 375-389, D. Reidel, Dordrecht, the Netherlands, 1988.

[KK89] E. KRANAKIS AND D. KRIZANC, *Computing Boolean functions on anonymous networks*, Centrum voor Wiskunde en Informatica, Tech. Report CS-8935, Amsterdam, the Netherlands, September 1989.

[LPS88] M. W. LIEBECK, C. E. PRAEGER, AND J. SAXL, *On the O'Nan-Scott theorem for finite primitive permutation groups*, J. Austral. Math. Soc. Ser. A, 44 (1988), pp. 389-396.

[LM85] E. M. LUKS AND P. MCKENZIE, *Fast parallel computation with permutation groups*, in Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985.

[MS78] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error Correcting Codes*, North-Holland, Amsterdam 1978.

[McC56] E. J. MCCLUSKEY, JR., *Detection of group invariance or total symmetry of a Boolean function*, Bell Systems Tech. J., 35 (1956), pp. 1445-1453.

[McK84] P. MCKENZIE, *Parallel complexity and permutation groups*, Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1984.

[MC85] P. MCKENZIE AND S. A. COOK, *The parallel complexity of abelian permutation group problems*, Tech. Report 181/85, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1985.

[Mul86] K. MULMULEY, *A fast parallel algorithm to compute the rank of a matrix over an arbitrary field*, in Proc. 18th ACM Symposium on Theory of Computing, 1986, pp. 338-339.

[Pas66] D. S. PASSMAN, *Permutation Groups*, W. A. Benjamin, New York, Amsterdam, 1966.

[Pip79] N. PIPPENGER, *On simultaneous resource bounds*, in Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 307-311.

[PR87] G. PÓLYA AND R. C. READ, *Combinatorial Enumeration of Groups, Graphs and Chemical Compounds*, Springer-Verlag, Berlin, New York, 1987.

[Sav76] J. E. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.

[Sha49] C. SHANNON, *The synthesis of two-terminal switching circuits*, Bell Systems Tech. J. (1949), pp. 59-98.

[SV84] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409-422.

[Tzu82] T. TSUZUKU, *Finite Groups and Finite Geometries*, Cambridge University Press, Cambridge, U.K., 1982.

[Wie64] H. WIELANDT, *Finite Permutation Groups*, Academic Press, New York, 1964.

[Yab83] S. YABLONSKY, *Introduction aux mathématiques discrètes*, MIR, (translated from the Russian), 1983.

[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. 20, No. 3, pp. 591–602, June 1991

© 1991 Society for Industrial and Applied Mathematics
012

TESTS FOR PERMUTATION POLYNOMIALS*

JOACHIM VON ZUR GATHEN†

Abstract. If $\mathbb{F}_q$ is a finite field and $f \in \mathbb{F}_q[x]$, then f is called a *permutation polynomial* if the mapping $\mathbb{F}_q \to \mathbb{F}_q$ induced by f is bijective. This property can be tested by a probabilistic algorithm whose number of operations is polynomial (in fact, essentially linear) in the input size, i.e., in $\deg f \cdot \log q$. This is extended to "almost permutation polynomials," whose value set consists of almost all elements of $\mathbb{F}_q$.

Key words. permutation polynomials, values of polynomials, finite fields, Euclidean remainder sequence, subresultant, probabilistic algorithm

AMS(MOS) subject classifications. 11T06, 12Y05, 68Q40

1. Introduction. A univariate polynomial $f \in \mathbb{F}_q[x]$ over a finite field $\mathbb{F}_q$ with q elements (q a power of a prime number) induces a function $\mathbb{F}_q \to \mathbb{F}_q$ via $a \mapsto f(a)$. If this function is bijective, then f is called a *permutation polynomial.* Permutation polynomials have been studied since Hermite [14] and Dickson [9], and recent interest stems from potential applications in public-key cryptography (see Lidl and Mullen [18]); reference to other uses is given in the latter article. A list of all permutation polynomials of degree at most 5 is given in Dickson [10] and Lidl and Niederreiter [19]. We may always assume, without loss of generality, that $\deg f < q$.

Given an arbitrary polynomial $f \in \mathbb{F}_q[x]$ of degree n, one can test whether it is a permutation polynomial simply by producing its list of values (see §2). Another general test goes back to Hermite and Dickson (see § 3). In their survey paper, Lidl and Mullen [18] pose as an open problem:

(P1) Find an algorithm of lower complexity than $O(qn)$ to test whether a given polynomial is a permutation polynomial of $\mathbb{F}_q$.

For such a test, the input size—the number of bits required to represent f—is about $n \log q$. The above-mentioned tests use exponential time, for large q, and no polynomial-time tests are in the literature. Lidl and Mullen [18] quote some criteria in terms of the coefficients of f. We present a probabilistic test whose number of operations in $\mathbb{F}_q$ is essentially $O(n \log q)$, i.e., essentially linear in the input size $n \log q$.

In §2, we briefly consider the "simple" test and find that off-the-shelf techniques from computer algebra already improve the running time slightly, without any new insights into the problem. Hermite's classical test has been one of the most important tools in the study of permutation polynomials, both for theoretical and practical purposes. Section 3 gives a probabilistic variant of this test, reducing the running time from $\Omega(q^2)$ to essentially $O(q)$. In §4, we derive a criterion saying that f is a permutation polynomial if and only if $g_f = 0$, where $g_f \in \mathbb{F}_q[y]$ is a new polynomial

* Received by the editors May 18, 1989; accepted for publication (in revised form) August 31, 1990. Part of this work was done while the author was a Visiting Fellow at the Computer Science Laboratory, Australian National University, Canberra, Australia, and supported by Natural Sciences and Engineering Research Council of Canada grant A-2514. A first version appeared as Tech. Report TR-CS-89-08, Computer Sciences Laboratory, Australian National University, and extended abstracts of partial results appeared in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 88–92, and Proc. International Symposium on Symbolic and Algebraic Computation, Tokyo, Japan, Association for Computing Machinery Press, 1990, pp. 140–144.

† Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A4, Canada (gathen@theory.toronto.edu).

whose coefficients are polynomials in the coefficients of f. This criterion is equivalent to one given by Raussnitz [23]. The main result of this paper is in §5, where we show how to calculate $g_f(u)$ fast for randomly chosen u in some finite extension field of $\mathbb{F}_q$. The resulting polynomial-time probabilistic test always gives the correct answer if the input is a permutation polynomial. If it is not, it may give the incorrect answer, but with controllably small probability ϵ. The running time is essentially proportional to $\log \epsilon^{-1}$. If we use $\epsilon = q^{-1}$, the running time is $O(n \log q)$, up to factors $\log n$: *softly linear* running time.

Precious few classes of permutation polynomials are known (see Lidl and Mullen [18]), and a random polynomial in $\mathbb{F}_q[x]$ of degree less than q is a permutation polynomial with very small probability $q!/q^q \approx e^{-q}$. (Recall that the polynomials of degree less than q correspond bijectively to the functions $\mathbb{F}_q \to \mathbb{F}_q$.)

To enlarge the pool of candidate polynomials, we generalize the notion of permutation polynomial as follows. Suppose some $\rho \in \mathbb{N}$ is given, and let $V(f) = \#f(\mathbb{F}_q)$ be the size of the image of f. We say that f is *ρ-large* if the image of the mapping f has at least $q - \rho$ elements: $V(f) \geq q - \rho$. Thus f is 0-large if and only if f is a permutation polynomial.

Section 6 gives a criterion for ρ-large polynomials analogous to the criterion in §4 for permutation polynomials, and §7 gives the resulting test. It is a probabilistic algorithm with expected time polynomial in $n\rho \log q$; in fact, the time is softly linear in $n\rho \log q$.

The method presented here suggests the following general question: which (special) problems can one solve in (random) polynomial time for polynomials of exponential degree given by small arithmetic circuits? Section 8 briefly discusses this.

A "naive" test for permutation polynomials is to choose some elements $u \in \mathbb{F}_q$ at random and check whether each has exactly one preimage under f. At first sight, it looks as if this test has little chance of success, e.g., for a polynomial whose values leave out only very few elements of $\mathbb{F}_q$. However, a geometric study of permutation polynomials, initiated by Hayes [13], leads to the essentially equivalent notion of *exceptional polynomials.* This property can also be tested in random polynomial time, and the approach shows that the above naive test has a good chance of success. Its running time is about the square of the time for the algorithm presented here (von zur Gathen [12]). Shparlinskiy [26] presents a deterministic test using essentially $O(n^3 q^{1/2})$ operations.

2. The simple test revisited. Given $f \in \mathbb{F}_q[x]$ of degree n, one can produce its list of values and sort them, to determine whether f is a permutation polynomial. This takes $O(nq)$ arithmetic operations, plus $O(q \log^2 q)$ binary operations for sorting. Alternatively, one can test whether the q values are distinct with $O(q \log^2 q \log\log q)$ arithmetic operations (Baur and Strassen [3]).

Since we know what the q values have to be, we can do better by checking the condition

$$\prod_{v \in \mathbb{F}_q} (x - f(v)) = x^q - x,$$

which is equivalent to f being a permutation polynomial. All $f(v)$ can be computed in $O(q \log^2 n \log\log n)$ arithmetic operations, and the product can be calculated at the same cost (see Borodin and Munro [4]).

3. Hermite's test revisited. Hermite's criterion says that $f \in \mathbb{F}_q[x]$ is a permutation polynomial if and only if

(i) f has exactly one root in $\mathbb{F}_q$,
(ii) $\forall i, 1 \leq i \leq q-2$, $\deg(f^i \text{ rem } (x^q - x)) \leq q-2$
(Lidl and Niederreiter [19, Thm. 7.4]). Here, $(g \text{ rem } h) \in \mathbb{F}_q[x]$ is the remainder of g on division by h: $(g \text{ rem } h) \equiv g \bmod h$ and $\deg(g \text{ rem } h) < \deg h$ (assuming $h \neq 0$). The obvious implementation of Hermite's test requires about q multiplications of f with a polynomial of degree less than q, each followed by a reduction modulo $x^q - x$. Even when $n = \deg f$ is small, say constant, this may require $\Omega(q^2)$ operations in $\mathbb{F}_q$. We now implement this test more efficiently.

With a new indeterminate y, we have

$$\begin{aligned}(f+y)^{q-1} &= \sum_{0 \leq i \leq q-1} \binom{q-1}{i} y^{q-1-i} f^i \\ &\equiv y^{q-1} + r_{q-1} + \sum_{1 \leq i \leq q-2} \binom{q-1}{i} y^{q-1-i} r_i \bmod x^q - x,\end{aligned}$$

where $r_i = f^i \text{ rem } x^q - x$ for $1 \leq i < q$. Each of these binomial coefficients is nonzero in $\mathbb{F}_q$ (Lucas [20]; Lidl and Niederreiter [19, Exercise 7.1]). Let

$$r = ((f+y)^{q-1} - f^{q-1}) \text{ rem } (x^q - x) \in \mathbb{F}_q[x, y],$$

and $s \in \mathbb{F}_q[y]$ be the coefficient of x^{q-1} in r. Then we have

$$\text{(ii)} \iff s = 0.$$

Since $\deg_y (f+y)^{q-1} = q-1$, we have $\deg_y s \leq q-1$. Computing $(f+y)^{q-1} \text{ rem } (x^q - x)$ as a bivariate polynomial would again result in cost $\Omega(q^2)$. However, we can substitute a randomly chosen $u \in \mathbb{F}_{q^m}$ for y, from a suitable extension $\mathbb{F}_{q^m}$ of $\mathbb{F}_q$, and compute

$$r(u) = ((f+u)^{q-1} - f^{q-1}) \text{ rem } (x^q - x) \in \mathbb{F}_{q^m}[x],$$

and $s(u)$ as the coefficient of x^{q-1} in $r(u)$. We return "YES" if $s(u) = 0$, and "NO" otherwise. (We also check condition (i): $\deg \gcd(x^q - x, f) = 1$.)

To estimate the cost, let $M : \mathbb{N} \to \mathbb{R}$ denote a "universal" cost of multiplication, i.e., let it be such that two polynomials of degree at most n over a ring R can be multiplied in $O(M(n))$ arithmetic operations in R, and two n-bit integers can be multiplied with $O(M(n))$ bit operations. We can choose $M(n) = n \log n \log\log n$ (Schönhage and Strassen [24], Cantor and Kaltofen [7]). If $g, h \in \mathbb{F}_q[x]$ are polynomials of degree at most n, then the division with remainder of g by h (if $h \neq 0$) can be performed in $O(M(n))$ operations in $\mathbb{F}_q$.

PROPOSITION 1. *The probabilistic algorithm given above can be implemented with $O(M(q) \log q \cdot M(m))$ arithmetic operations in $\mathbb{F}_q$. Its output is correct with probability at least $1 - q^{1-m}$.*

Proof. The algorithm can be performed in $O(M(q) \log q)$ operations in $\mathbb{F}_{q^m}$, using "repeated squaring." Elements of $\mathbb{F}_{q^m}$ are represented by their coordinates in $(\mathbb{F}_q)^m$, and a single arithmetic operation on such elements can be performed with $M(m)$ operations in $\mathbb{F}_q$. Finally, the gcd condition can be checked with $O(\log q M(n))$ operations in $\mathbb{F}_q$ (Aho, Hopcroft, and Ullman [1, §8.9]). Thus the total cost is $O(M(q) \log q \cdot M(m))$ operations in $\mathbb{F}_q$. (We have neglected the cost of constructing $\mathbb{F}_{q^m}$; see §5.)

Assume that (i) holds. If f is a permutation polynomial, then $s = 0$ and $s(u) = 0$. If f is not a permutation polynomial, then $s \neq 0$ and $s(u) = 0$ occurs with probability at most $\deg s / q^m < q^{1-m}$. $\square$

We can make the error probability arbitrarily small, by choosing m appropriately. The running time of this test is still at least linear in q. However, the idea of random substitutions will lead to a substantial improvement in §5.

In practical applications of Hermite's test, one stops of course as soon as $\deg r_i = q-1$ is found for some $i \le q-2$. In the same vein, our algorithm would first calculate f^{q-1} rem $x^q - x$ by repeated squaring, and stop whenever one of the (few) powers of $f(\text{rem } x^q - x)$ calculated is found to have degree $q-1$. One proceeds similarly in the subsequent computation of certain $\left((f+u)^i - f^i\right)$ rem $x^q - x$ by repeated squaring.

4. A criterion for permutation polynomials. By definition, a polynomial $f \in \mathbb{F}_q[x]$ is a permutation polynomial if and only if for all $u \in \mathbb{F}_q$ there exists a unique $v \in \mathbb{F}_q$ with $f(v) = u$, or, equivalently, $x - v$ divides $f - u$ in $\mathbb{F}_q[x]$. Since $\mathbb{F}_q$ is a finite set, surjectivity is sufficient:

$$\begin{array}{rl} f & \text{is a permutation polynomial} \\ \iff & \forall u \in \mathbb{F}_q\ \exists v \in \mathbb{F}_q \quad f(v) = u \\ \iff & \forall u \in \mathbb{F}_q\ \exists v \in \mathbb{F}_q \quad x - v \mid f - u. \end{array}$$

For two nonzero polynomials $a = a_m x^m + \cdots + a_0$, $b = b_n x^n + \cdots + b_0 \in \mathbb{F}_q[x]$, we denote by $\gcd(a,b)$ their *monic* gcd. If $a_m b_n \neq 0$, then

$$R(a,b) = \left|\begin{array}{cccccccc} a_m & a_{m-1} & \cdots & a_0 & & & & \\ & a_m & a_{m-1} & \cdots & a_0 & & & \\ & & \ddots & \ddots & & \ddots & & \\ & & & a_m & a_{m-1} & \cdots & \cdots & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_0 & & & \\ & b_n & b_{n-1} & \cdots & \cdots & b_0 & & \\ & & \ddots & \ddots & & & \ddots & \\ & & & b_n & b_{n-1} & \cdots & \cdots & b_0 \end{array}\right| \in \mathbb{F}_q^{(m+n)\times(m+n)}$$

is their Sylvester matrix, consisting of n rows of coefficients of a, and m rows of coefficients of b. Furthermore, $\text{res}(a,b) = \det R(a,b)$ is their resultant. A fundamental fact is that

$$\gcd(a,b) \neq 1 \iff \text{res}(a,b) = 0$$

(see van der Waerden [28]). Since $x^q - x = \prod_{b \in \mathbb{F}_q}(x - b)$, we have

$$\begin{array}{rl} f & \text{is a permutation polynomial} \\ \iff & \forall u \in \mathbb{F}_q\ \gcd(x^q - x, f - u) \neq 1 \\ \iff & \forall u \in \mathbb{F}_q\ \text{res}(x^q - x, f - u) = 0 \\ \iff & y^q - y \mid \text{res}(x^q - x, f - y). \end{array}$$

Here, y is a new indeterminate, $h_f = \text{res}(x^q - x, f - y) \in \mathbb{F}_q[y]$, and the divisibility condition is in $\mathbb{F}_q[y]$.

What are the degree and leading coefficient of h_f? The constant term $f(0) - y$ of $f - y \in \mathbb{F}_q[y][x]$ occurs q times on the lower right part of the diagonal of $R(x^q - x, f - y)$, and nowhere else. The cofactor in the Laplace expansion for the determinant of that part of the diagonal is the upper left $n \times n$-submatrix, which is upper triangular with 1's on the diagonal, and has determinant 1. Thus h_f has degree exactly q,

and leading coefficient $(-1)^q$. It follows that the condition "$y^q - y \mid h_f$" is equivalent to "$h_f = (-1)^q(y^q - y)$." We have proved the following criterion for permutation polynomials.

THEOREM 2. *Let $f \in \mathbb{F}_q[x]$, and*

$$g_f = \operatorname{res}(x^q - x, f - y) - (-1)^q(y^q - y) \in \mathbb{F}_q[y].$$

Then f is a permutation polynomial if and only if $g_f = 0$.

Easy matrix manipulations show that Raussnitz's criterion [23], when expressed in terms of the Sylvester matrix, is equivalent to the above.

5. Testing permutation polynomials. Let $f \in \mathbb{F}_q[x]$ have degree n. We want to use Theorem 2 to test efficiently whether f is a permutation polynomial or not. Computing the resultant as the determinant of the $(q+n) \times (q+n)$-Sylvester matrix would be very costly. We can, however, use the Euclidean algorithm to compute the resultant as follows.

Let F be any field, $a_0, a_1 \in F[x]$ of degrees $n_0 \geq n_1 \geq 0$, respectively, and consider the Euclidean scheme for (a_0, a_1), consisting of the remainders $a_0, a_1, a_2, \cdots, a_l \in F[x]$ and the quotients $q_1, \cdots, q_l \in F[x]$ in the Euclidean algorithm for a_0 and a_1, defined by

$$a_{i-1} = q_i a_i + a_{i+1} \text{ and } \deg a_{i+1} < \deg a_i \tag{1}$$

for $1 \leq i \leq l$, using $a_{l+1} = 0$. This scheme always exists and is unique. $(q_1, \cdots, q_l, a_l)$ is called the *Euclidean representation* of (a_0, a_1) (Knuth [17] and Strassen [27]). Furthermore, let $n_i = \deg a_i$, $d_i = \deg q_i$, $\alpha_i \in F$ be the leading coefficient of a_i, and $\gamma_i \in F$ the leading coefficient of q_i. The "fundamental theorem on polynomial remainder sequences" says that if $n_l \geq 1$, then $\operatorname{res}(a_0, a_1) = 0$, and if $n_l = 0$, then

$$\operatorname{res}(a_0, a_1) = (-1)^s \alpha_l^{n_{l-1}} \prod_{1 \leq i < l} \alpha_i^{n_{i-1} - n_{i+1}}, \tag{2}$$

where $s = \sum_{0 \leq i < l} n_i n_{i+1}$ (Collins [8] and Brown and Traub [6]).

Equation (1) implies that $\alpha_{i-1} = \gamma_i \alpha_i$ and $n_{i-1} = d_i + n_i$ for all i. It follows that

$$\alpha_i = \alpha_1 \prod_{2 \leq j \leq i} \gamma_j^{-1}$$

for $2 \leq i \leq l$. Substituting this into (2) and collecting powers of γ_i, we find

$$\operatorname{res}(a_0, a_1) = (-1)^s \alpha_1^{n_0 + n_1} \prod_{2 \leq i \leq l} \gamma_i^{-(n_{i-1} + n_i)}, \tag{3}$$

if $n_l = 0$.

Thus we could calculate our g_f by executing the Euclidean algorithm for $x^q - x$ and $f - y$ in $\mathbb{F}_q(y)[x]$. Again, this would be very inefficient since the first division of $x^q - x$ by $f - y$ may already leave us with a remainder whose degree in y is very large—about q/n. We circumvent this problem by substituting a random element $u \in \mathbb{F}_{q^m}$ for y, using an appropriate extension $\mathbb{F}_{q^m}$ of $\mathbb{F}_q$.

ALGORITHM TEST FOR PERMUTATION POLYNOMIAL.

Input: Coefficients of a monic polynomial $f \in \mathbb{F}_q[x]$ of degree n, $2 \le n < q$, and confidence parameter $\epsilon > 0$. [Intuitively, $\epsilon \ll 1$. Note that any linear polynomial $ax + b$ with $a \ne 0$ is a permutation polynomial.]

Output: YES or NO.

1. Set $m = 1 + \lceil \log_q(2\epsilon^{-1}) \rceil$, and find an irreducible polynomial $\varphi \in \mathbb{F}_q[z]$ of degree m.
2. Choose (uniformly) a random element $u \in \mathbb{F}_{q^m} = \mathbb{F}_q[z]/(\varphi)$.
3. Set $a_0 = x^q - x$, $a_1 = f - u$, and compute the coefficients of $a_2 = (x^q - x) \text{ rem } (f - u) \in \mathbb{F}_{q^m}[x]$. This division with remainder is performed by "repeated squaring" of x, reducing modulo $f - u$ after each multiplication step.
4. Compute the Euclidean representation $(q_2, \cdots, q_l, a_l)$ for (a_1, a_2) in $\mathbb{F}_{q^m}[x]$, let $d_i = \deg q_i$ and $\gamma_i \in \mathbb{F}_{q^m}$ be the leading coefficient of q_i, for $2 \le i \le l$. If $\deg a_l \ge 1$, return YES and stop.
5. Compute $n_0 = q$, $n_1 = n$, and $n_2, \cdots, n_l$ from $n_i = n_{i-1} - d_i$, and calculate $s = \sum_{0 \le i < l} n_i n_{i+1} \text{ rem } 2$.
6. Compute

$$v = (-1)^s \prod_{2 \le i \le l} \gamma_i^{-(n_{i-1}+n_i)} - (-1)^q (u^q - u) \in \mathbb{F}_{q^m}.$$

7. Return YES if $v = 0$, and NO otherwise. □

It is convenient to ignore logarithmic factors using the "soft O" notation, introduced by von zur Gathen [11] and Babai, Luks, and Seress [2]:

$$g = O\tilde{}(h) \iff \exists k \; g = O(h(\log_2 h)^k).$$

THEOREM 3. *The algorithm can be performed with m random choices in $\mathbb{F}_q$, plus the cost of finding an irreducible polynomial of degree m, and $O(\log q \cdot M(n)M(m))$ arithmetic operations in $\mathbb{F}_q$, where $m = 1 + \lceil \log_q(2\epsilon^{-1}) \rceil$. These are $O\tilde{}(n \log_2 \epsilon^{-1})$ operations in $\mathbb{F}_q$ if $\epsilon \le q^{-1}$. If f is a permutation polynomial, the output is* YES. *If f is not a permutation polynomial, the output is* NO *with probability at least $1 - \epsilon$.*

Proof. Step 3 can be done in $O(\log q \cdot M(n))$ operations in $\mathbb{F}_{q^m}$. The usual algorithm for the Euclidean scheme calculates all quotients and remainders in $O(n^2)$ arithmetic operations. However, the Euclidean representation $(q_2, \cdots, q_l, a_l)$ can be computed in only $O(M(n) \cdot \log n)$ arithmetic operations by the Knuth–Schönhage algorithm (see Aho, Hopcroft, and Ullman [1, §8.9] and Strassen [27]). Each $\gamma_i^{n_{i-1}+n_i}$ can be calculated in $O(\log n)$ operations, and thus step 6 requires $O(l \log n + \log q)$ or $O(n \log n + \log q)$ operations in $\mathbb{F}_{q^m}$. Since one operation in $\mathbb{F}_{q^m}$ can be simulated with $O(M(m))$ operations in $\mathbb{F}_q$, the total cost is $O(\log q \cdot M(n)M(m))$ operations in $\mathbb{F}_q$. This is $O\tilde{}(n \log \epsilon^{-1})$ if $\epsilon \le q^{-1}$, since then $\log_2 q \cdot \lceil \log_q(2\epsilon^{-1}) \rceil = O(\log_2 \epsilon^{-1})$.

Set $A_1 = f - y \in \mathbb{F}_q(y)[x]$. Then $h_f = \text{res}_x(a_0, A_1)$. Since $a_1 = A_1(u) \in \mathbb{F}_{q^m}[x]$ has the same degree as A_1, we have $\text{res}_x(a_0, a_1) = h_f(u)$ and $v = g_f(u)$ in step 6. If f is a permutation polynomial, then $g_f = 0$ and $v = 0$. If f is not a permutation polynomial, then $g_f \in \mathbb{F}_q[y]$ is a nonzero polynomial of degree less then q, and $v \ne 0$ with probability more than $1 - q/q^m \ge 1 - \epsilon/2$.

If $\deg a_l \ge 1$ in step 4, then $f(v) = u$ for some $v \in \mathbb{F}_q$, and thus $u \in \mathbb{F}_q$; the probability of this happening is at most $q/q^m \le \epsilon/2$. □

If we choose ϵ polynomial in q^{-1}, then the algorithm uses $O\tilde{~}(n \log q)$ operations. Some Boolean operations occur in the algorithm, e.g., in the calculation of m, the n_i, and s; we have neglected the small cost of these.

We have not specified a method for finding an irreducible polynomial in step 1. Rabin [22] gives a probabilistic algorithm for this problem, using $O(kmM(m) \log m \log q)$ operations in $\mathbb{F}_q$, and returning successfully with probability at least $1 - m^{-k}$, for any k. Choosing $k = \log \epsilon^{-1}$ gives failure probability at most ϵ for this step, and cost $O(\log \epsilon^{-1} mM(m) \log m \log q)$, which is $O\tilde{~}(\log n \log^3 \epsilon^{-1})$ if $\epsilon \leq q^{-1}$.

We actually do not need φ of degree exactly m, but degree between m and $2m$, say, is sufficient. This may be useful if a table of irreducible polynomials is available, or if a particular degree is preferable, say powers of 2. One could even take a random $\varphi \in \mathbb{F}_q[z]$ of degree m without linear factors (i.e., $\gcd(z^q - z, \varphi) = 1$) and compute in the "pretend-field" $\mathbb{F}_q[z]/(\varphi)$; if a division by a zero-divisor turns up in the algorithm, this gives a factorization $\varphi = \varphi_1 \cdot \varphi_2$, and one can continue in the two rings $\mathbb{F}_q[z]/(\varphi_1)$ and $\mathbb{F}_q[z]/(\varphi_2)$. The algorithm would still work (by the Chinese Remainder Theorem), but the analysis is slightly more complicated.

Yet another possibility would be to take φ of degree 2 if $n \leq q/2$ (or n is not too close to q), and of degree 3 otherwise; run the algorithm with several random choices $u_1, \cdots, u_{2k+1}$ in $\mathbb{F}_{q^2}$ (respectively, $\mathbb{F}_{q^3}$); and take a majority vote on the individual outcomes. Each individual run has an error probability at most qn/q^2 (respectively, qn/q^3), and for $k = \log_2 \epsilon^{-1}$ (respectively, $k = \log_q \epsilon^{-1}$), the total error probability is at most ϵ. The cost of this implementation is $O\tilde{~}((n + \log q) \log \epsilon^{-1})$ (respectively, $O\tilde{~}(n \log \epsilon^{-1})$), plus the cost of finding φ.

Instead of studying polynomials inducing permutations on $\mathbb{F}_q$, Brawley, Carlitz, and Levine [5] consider permutations of the matrix algebra $\mathbb{F}_q^{d \times d}$, and prove that $f \in \mathbb{F}_q[x]$ is a permutation polynomial of $\mathbb{F}_q^{d \times d}$ if and only if f is a permutation polynomial of $\mathbb{F}_q, \mathbb{F}_{q^2}, \cdots, \mathbb{F}_{q^d}$, and the derivative f' does not have a root in $\mathbb{F}_q, \mathbb{F}_{q^2}, \cdots, \mathbb{F}_{q^k}$, where $k = \lfloor d/2 \rfloor$.

COROLLARY 4. *Let $f \in \mathbb{F}_q[x]$ have degree n, and $0 < \epsilon \leq q^{-d}$. There is a probabilistic algorithm which determines whether f is a permutation polynomial on $\mathbb{F}_q^{d \times d}$ correctly with probability at least $1 - \epsilon$. Apart from random choices and the finding of certain irreducible polynomials, it uses $O\tilde{~}(d \log \epsilon^{-1}(n/\log q + d))$, or $O\tilde{~}(d^2 n \log \epsilon^{-1})$, operations in $\mathbb{F}_q$.*

Proof. We simply implement the first of the Brawley, Carlitz, and Levine conditions using the Algorithm Test for Permutation Polynomial, with

$$O\tilde{~}((n + j \log q) \log_{q^j}(n/\epsilon))$$

arithmetic operations in $\mathbb{F}_{q^j}$, for $1 \leq j \leq d$, each costing $O\tilde{~}(j)$ operations in $\mathbb{F}_q$. This leads to the stated bound. The second condition

$$\gcd(x^{q^j} - x, f') = 1 \quad \text{for } 1 \leq j \leq k$$

can also be tested at this cost. □

6. A criterion for large polynomials. Let $\rho \in \mathbb{N}$, and recall the notion of ρ-large from the introduction, and $h_f \in \mathbb{F}_q[y]$ from §4. Then we have:

$$\begin{aligned} f \text{ is } \rho\text{-large} &\iff \exists P \subseteq \mathbb{F}_q \ (\#P \geq q - \rho \text{ and } \forall u \in P \ \exists v \in \mathbb{F}_q \ f(v) = u) \\ &\iff \exists P \subseteq \mathbb{F}_q \ (\#P \geq q - \rho \text{ and } \forall u \in P \ \exists v \in \mathbb{F}_q \ x - v \mid f - u) \\ &\iff \exists P \subseteq \mathbb{F}_q \ (\#P \geq q - \rho \text{ and } \forall u \in P \ \gcd(x^q - x, f - u) \neq 1) \end{aligned}$$

$$\begin{aligned}
&\iff \exists P \subseteq \mathbb{F}_q \ (\#P \geq q-\rho \text{ and } \forall u \in P \ \ h_f(u) = 0) \\
&\iff \exists P \subseteq \mathbb{F}_q \ (\#P \geq q-\rho \text{ and } \forall u \in P \ \ y-u \mid h_f) \\
&\iff \deg(\gcd(y^q - y, h_f)) \geq q-\rho \\
&\iff \deg((y^q - y)/k_f) \leq \rho \\
&\iff \deg(h_f/k_f) \leq \rho,
\end{aligned}$$

where $k_f = \gcd(y^q - y, h_f) \in \mathbb{F}_q[y]$. Thus we have the following criterion for ρ-large polynomials.

THEOREM 5. *Let $f \in \mathbb{F}_q[x]$, and $h_f = \text{res}_x(x^q - x, f-y)$ and $k_f = \gcd(y^q - y, h_f)$ in $\mathbb{F}_q[y]$. Then f is ρ-large if and only if $\deg((y^q - y)/k_f) \leq \rho$.*

If f is ρ-large, then all but at most 2ρ elements u of $\mathbb{F}_q$ have exactly one preimage v under f; this unique v can be easily found from $x - v = \gcd(x^q - x, f - u)$.

7. A test for large polynomials. Let $a_0 = x^q - x$, $A_1 = f - y \in \mathbb{F}_q(y)[x]$, and $Q_1, \cdots, Q_l, A_l \in \mathbb{F}_q(y)[x]$ be the Euclidean representation of (a_0, A_1) (i.e., the quotients and the gcd as calculated by the Euclidean algorithm). If $u \in \mathbb{F}_{q^m}$, $a_1 = A_1(u) = f - u \in \mathbb{F}_{q^m}[x]$, and $q_1, \cdots, q_{l'}, a_{l'}$ is the Euclidean representation of (a_0, a_1) in $\mathbb{F}_{q^m}[x]$, then $l = l'$, $q_i = Q_i(u)$ for all i, and $a_l = A_l(u)$. In particular, $\deg q_i = \deg Q_i$.

The Euclidean algorithm for (a_0, A_1) requires, of course, tests for zero or branching, in order to determine the degree sequence $(\deg Q_1, \cdots, \deg Q_l, \deg A_l)$. However, the computation using u gives us the correct degree sequence, and then all entries of the Euclidean scheme are rational functions in the coefficients of a_0 and A_1; the subresultant theory provides explicit formulas. In fact, we obtain an *arithmetic circuit* (or straight-line program) for h_f, i.e., a computation using only the coefficients of f and the operations $+, -, *, /$. The size of an arithmetic circuit is the number of arithmetic operations in it.

It is not clear how to calculate efficiently $k_f = \gcd(y^q - y, h_f)$, if we regard the degree q of $B_1 = y^q - y$ and $B_2 = h_f$ as exponentially large. However, in

$$B_0 = \frac{B_1}{B_2} = \frac{y^q - y}{h_f} = \frac{(y^q - y)/k_f}{h_f/k_f} = \frac{C_1}{C_2}$$

the two polynomials $C_1 = (y^q - y)/k_f$ and $C_2 = h_f/k_f$ are relatively prime. We can now call Kaltofen's [15] Algorithm Rational Numerator and Denominator, to calculate those two polynomials from the arithmetic circuits for h_f (discussed above) and $y^q - y$ (repeated squaring).

FACT 6 (Kaltofen [15]). *Suppose an arithmetic circuit α of size s with one input y over a field F computes $B_0 = C_1/C_2 \in F(y)$, with $C_1, C_2 \in F[y]$ relatively prime, and that $u \in F$ is such that no division by zero occurs in α on input $y \leftarrow u$. Then, given α and u, and an integer ρ, one can compute (deterministically) with $O(M(\rho)(s + \log \rho))$ arithmetic operations in F an arithmetic circuit β of size $O(M(\rho)(s + \log \rho))$ over F which computes two polynomials c_1 and c_2 in $F[y]$ of degree at most ρ such that if $\deg C_1, \deg C_2 \leq \rho$, then $c_1 = C_1$ and $c_2 = C_2$. β has no divisions by zero on input $y \leftarrow u$.*

This is a special case of Kaltofen's theorem 8.1 [15]. To derive it, we note that we can replace Kaltofen's "Step FT" with $a_1 = u$, and all the nasty possibilities that complicate Kaltofen's proof (for multivariate polynomials) vanish in our simple case. In particular, this variant is deterministic, while in the general case, Kaltofen needs probabilistic choice. Note that, if $\deg C_1 > \rho$ or $\deg C_2 > \rho$, then $B_0 \neq c_1/c_2$. (We

ignore the Boolean cost of the procedure, and assume, also in the sequel, a reasonable convention for $\rho = 0$ in the O-notation.)

The following algorithm results from the above discussion.

ALGORITHM TEST FOR POLYNOMIAL WITH LARGE IMAGE.

Input: Coefficients of a monic polynomial $f \in \mathbb{F}_q[x]$ of degree n, $2 \le n < q$, some $\rho \in \mathbb{N}$ with $0 \le \rho \le q$, and a confidence parameter $\epsilon > 0$.

Output: YES or NO.

i. Set $m = 1 + \lceil \log_q(3\epsilon^{-1}) \rceil$, and find an irreducible polynomial $\varphi \in \mathbb{F}_q[z]$ of degree m.

ii. Perform steps 2, 3, 4, and 5 of Algorithm Test for Permutation Polynomial.

iii. Compute $b_2 = (-1)^s \prod_{2 \le i \le l} \gamma_i^{-(n_{i-1}+n_i)} \in \mathbb{F}_{q^m}$, and $b_1 = u^q - u$.

iv. Let $A_1 = f - y \in \mathbb{F}_q(y)[x]$. Consider the following arithmetic circuit α over $\mathbb{F}_q$ with one input y, working in five stages.

a. Compute $A_2 \equiv x^q - x \bmod A_1$.

b. Compute the "Euclidean representation" $(Q_2, \cdots, Q_l, A_l)$ for (A_1, A_2) in $\mathbb{F}_q(y)[x]$, using the degree sequence $(d_2, \cdots, d_l)$ computed in step ii.

c. Let $\Gamma_i \in \mathbb{F}_q(y)$ be the leading coefficient of Q_i, and compute

$$B_2 = h_f = (-1)^s \prod_{2 \le i \le l} \Gamma_i^{-(n_{i-1}+n_i)} \in \mathbb{F}_q[y].$$

d. Compute $B_1 = y^q - y$, by repeated squaring.

e. Compute the output $B_0 = B_1/B_2$.

[Note that in this step we do not actually calculate B_0, but rather describe an arithmetic circuit for B_0.]

v. Call Kaltofen's Algorithm Rational Numerator and Denominator (Fact 6) with input α and u, and degree bound ρ both for numerator and denominator. The output is an arithmetic circuit β computing two polynomials c_1 and c_2 in $\mathbb{F}_q[y]$ of degree at most ρ. [If $\deg C_1 \le \rho$, with $C_1 = B_1/k_f$ as above, then $B_0 = c_1/c_2$.]

vi. Execute β with input u to calculate $c_1(u)$ and $c_2(u)$, and compute $c_3 = b_1 \cdot c_2(u) - b_2 \cdot c_1(u)$. If $c_3 = 0$, then output YES; otherwise output NO.

THEOREM 7. *The algorithm can be performed with* $m = 1 + \lceil \log_q(3\epsilon^{-1}) \rceil$ *random choices in* $\mathbb{F}_q$, *plus the cost of finding an irreducible polynomial of degree* m, *and*

$$O(M(m)M(n)M(\rho)\log q)$$

arithmetic operations in $\mathbb{F}_q$. *These are* $O\tilde{~}(n\rho \log_2 \epsilon^{-1})$ *operations if* $\epsilon \le q^{-1}$. *If* f *is* ρ*-large, the output is* YES. *If* f *is not* ρ*-large, the output is* NO *with probability at least* $1 - \epsilon$.

Proof. If $n_l \ge 1$ in step ii, then $\gcd(x^q - x, f - u) \ne 1$ and hence $f(v) = u$ for some $v \in \mathbb{F}_q$, and thus $u \in \mathbb{F}_q$. This occurs with probability at most $q/q^m \le \epsilon/3$.

The subresultant theory (Brown and Traub [6]) guarantees that h_f is correctly computed in step iv.c, by (3). Recall $B_0 = B_1/B_2 = C_1/C_2$, with $C_1, C_2 \in \mathbb{F}_q[y]$ relatively prime, and let $C_3 = C_1 \cdot c_2 - C_2 \cdot c_1 \in \mathbb{F}_q[y]$. If the correct output is YES, so that $\deg C_1, \deg C_2 \le \rho$, then Fact 6 says that $c_1 = C_1$ and $c_2 = C_2$. Then $C_3 = 0$ and $c_3 = C_3(u) = 0$, and the correct answer YES will be output.

If the correct output is NO, then we know that $\deg C_1$ and $\deg C_2$ are larger than ρ. Step v will output two polynomials c_1 and c_2 of degree at most ρ, essentially

unrelated to our problem. Now $C_3 \in \mathbb{F}_q[y]$ is nonzero of degree at most $q+\rho$, and an incorrect output in step vi implies that $C_3(u) = 0$, which happens with probability at most $(q+\rho)q^{-m} \le 2\epsilon/3$ (Schwartz [25]). The total error probability is at most $\epsilon/3 + 2\epsilon/3 = \epsilon$.

We assume the standard representation of elements of $\mathbb{F}_{q^m}$ by vectors in $\mathbb{F}_q^m$. The number of random choices in $\mathbb{F}_q$ is m. Using $n < q$, one finds that the size s of α is $s = O(M(n)\log q)$, and $\log \rho = O(s)$. The number of arithmetic operations in $\mathbb{F}_q$ is $O(sM(m))$ in steps i through iii, negligible in step iv (which just sets up a circuit), $O(sM(\rho))$ in step v, and $O(sM(m)M(\rho))$ in step vi. Thus the total number of arithmetic operations in $\mathbb{F}_q$ is $O(M(m)M(n)M(\rho)\log q)$, which is $O\tilde{}(n\rho\log_2 \epsilon^{-1})$ if $\epsilon \le q^{-1}$. $\square$

For $f \in \mathbb{F}_q[x]$, let $\rho_f = q - V(f)$, so that f is ρ_f-large and not $(\rho_f - 1)$-large. By a "binary search on ρ" one can compute ρ_f with $O\tilde{}(n\rho_f \log_2 \epsilon^{-1})$ operations correctly with probability at least $1-\epsilon$.

8. Manipulating polynomials of large degree. A central ingredient of the algorithms presented here are efficient computations (for special problems) with polynomials of large (exponential) degree and small (polynomial-size) arithmetic circuits. It remains open how to put this development into a more general framework. Suppose we have two polynomials f and g (over a field, in many variables), given by two arithmetic circuits of size s and t, respectively, and an integer ρ, not larger than 2^s and 2^t. Kaltofen's methods can decide, e.g., whether $\deg f \le \rho$ in random polynomial time $(\rho s)^{O(1)}$. If $\deg f = \deg g$ is known, then the method given above (namely, computing the reduced numerator and denominator of f/g) can decide whether $\deg\gcd(f,g) \ge \deg f - \rho$ in random polynomial time $(\rho st)^{O(1)}$; $\deg f$ may be exponentially large.

In fact, this method only requires an estimate $e \ge |\deg f - \deg g|$, and uses time $(\rho ste)^{O(1)}$. As an application, suppose that $\mathrm{char}(F) = 0$ and, for simplicity, that $f \in F[x]$. One can easily find a small arithmetic circuit for $f' = \partial f/\partial x$; Baur and Strassen [3] produce one of size at most $5s$ even in the multivariate case. Then $\deg f' = \deg f - 1$, and we can test in time $(\rho s)^{O(1)}$ whether $\deg\gcd(f,f') \ge \deg f - \rho$, i.e., whether the squarefree part of f has degree at most ρ.

Here is a list of a few problems in manipulation of polynomials of exponentially large degree that one would like to answer in time $(\rho st)^{O(1)}$. For some, $\deg f$ might be an additional input.

(1) Test whether $\deg f \le \rho$ in time polynomial in $\log \rho$.
(2) Does g divide f?
(3) Is f squarefree? Does the squarefree part of f have degree at least $(\deg f) - \rho$?
(4) Is $\deg\gcd(f,g) \le \rho$? (This is probably a difficult problem; Plaisted [21] shows that the question "is $\gcd(f,g) \ne 1$?" is NP-hard for $f, g \in \mathbb{Q}[x]$.)
(5) Can one compute the Euclidean representation of (f,g) in time polynomial in the input *plus output* size, say in the sparse representation? For this, it seems sufficient to have a (probabilistic) polynomial-time test for

$$\deg f \overset{?}{\le} d$$

(given f as above and the binary representation of $d \in \mathbb{N}$), due to the rational nature of the Euclidean scheme for fixed degree sequence.
(6) Do some (or all) irreducible factors of f have degree at most ρ? Degree at least $(\deg f) - \rho$?

The positive results mentioned above easily carry over to the "black box" model; Kaltofen and Trager [16] present the necessary (probabilistic) algorithms. For questions (2)–(6), one might start by considering the sparse representation of f.

Acknowledgments. I thank Rudolf Lidl for discussions during my visit to the University of Tasmania and for his continued support with pointers and references, Brendan McKay for help with some computations, and Victor Shoup for an improvement in the proof of Theorem 3.

REFERENCES

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

[2] L. BABAI, E. M. LUKS, AND Á. SERESS, *Fast management of permutation groups*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, White Plains, NY, 1988, pp. 272–282.

[3] W. BAUR AND V. STRASSEN, *The complexity of partial derivatives*, Theoret. Comput. Sci., 22 (1982), pp. 317–330.

[4] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[5] J.V. BRAWLEY, L. CARLITZ, AND J. LEVINE, *Scalar polynomial functions on the $n \times n$ matrices over a finite field*, Linear Algebra Appl., 10 (1975), pp. 199–217.

[6] W. S. BROWN AND J. F. TRAUB, *On Euclid's algorithm and the theory of subresultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 505–514.

[7] D. G. CANTOR AND E. KALTOFEN, *Fast multiplication of polynomials over arbitrary rings*, Tech. Report 87-35, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1987; Acta Inform., to appear.

[8] G. E. COLLINS, *The calculation of multivariate polynomial resultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 515–532.

[9] L.E. DICKSON, *The analytic representation of substitutions on a power of a prime number of letters with a discussion of the linear group*, Ann. of Math., 11 (1897), pp. 65–120, 161–183.

[10] ———, *Linear Groups with an Exposition of the Galois Field Theory*, Teubner, Leipzig, Stuttgart, 1901; Dover, New York, 1958.

[11] J. VON ZUR GATHEN, *Irreducibility of multivariate polynomials*, J. Comput. System Sci., 31 (1985), pp. 225–264.

[12] ———, *Values of polynomials over finite fields*, Bull. Austral. Math. Soc., 43 (1991), pp. 141–146.

[13] D. R. HAYES, *A geometric approach to permutation polynomials over a finite field*, Duke Math. J., 34 (1967), pp. 293–305.

[14] C. HERMITE, *Sur les fonctions de sept lettres*, C.R. Acad. Sci. Paris, 57 (1863), pp. 750–757; also in Œuvres, Vol. 2, Gauthier-Villars, Paris, 1908, pp. 280–288.

[15] E. KALTOFEN, *Greatest common divisors of polynomials given by straight-line programs*, J. Assoc. Comput. Mach., 35 (1988), pp. 231–264.

[16] E. KALTOFEN AND B. M. TRAGER, *Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators*, J. Symb. Comp., 9 (1990), pp. 301–320.

[17] D. E. KNUTH, *The analysis of algorithms*, in Proc. Internat. Congress of Mathematicians, Vol. 3, Nice, France, 1970, pp. 269–274.

[18] R. LIDL AND G.L. MULLEN, *When does a polynomial over a finite field permute the elements of the field?*, Amer. Math. Monthly, 95 (1988), pp. 243–246.

[19] R. LIDL AND H. NIEDERREITER, *Finite Fields*, Vol. 20, Encyclopedia of Mathematics and Its Applications, Addison–Wesley, Reading MA, 1983.

[20] E. LUCAS, *Sur les congruences des nombres eulériens et des coefficients différentiels des fonctions trigonométriques, suivant un module premier*, Bull. Soc. Math. France, 6 (1877/78), pp. 49–54.

[21] D. A. PLAISTED, *New NP-hard and NP-complete polynomial and integer divisibility problems*, Theoret. Comput. Sci., 31 (1984), pp. 125–138.

[22] M. O. RABIN, *Probabilistic algorithms in finite fields*, SIAM J. Comput., 9 (1980), pp. 273–280.

[23] G. RAUSSNITZ, *Zur Theorie der Congruenzen höheren Grades*, Math. Naturwiss. Ber. Ungarn, 1 (1883), pp. 266–278.

[24] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation großer Zahlen*, Computing, 7 (1971), pp. 281–292.

[25] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[26] I.E. SHPARLINSKIY, *Private communication*, August 1990.

[27] V. STRASSEN, *The computational complexity of continued fractions*, SIAM J. Comput., 12 (1983), pp. 1–27.

[28] B. L. VAN DER WAERDEN, *Algebra*, Vol. 1, Seventh Edition, Frederick Ungar, New York, 1970.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 603-621, August 1991

© 1991 Society for Industrial and Applied Mathematics
001

CONSTRUCTIVE WHITNEY-GRAUSTEIN THEOREM: OR HOW TO UNTANGLE CLOSED PLANAR CURVES*

KURT MEHLHORN† AND CHEE-KENG YAP‡

Abstract. The classification of polygons is considered in which two polygons are *regularly equivalent* if one can be continuously transformed into the other such that for each intermediate polygon, no two adjacent edges overlap. A discrete analogue of the classic Whitney-Graustein theorem is proven by showing that the winding number of polygons is a complete invariant for this classification. Moreover, this proof is constructive in that for any pair of equivalent polygons, it produces some sequence of *regular transformations* taking one polygon to the other. Although this sequence has a quadratic number of transformations, it can be described and computed in real time.

Key words. polygons, computational algebraic topology, computational geometry, Whitney-Graustein theorem, winding number

AMS(MOS) subject classifications. 68Q20, 55M25

1. Why a circle differs from a figure-of-eight.

First consider closed planar curves that are smooth. Intuitively, a "kink" on such a curve is a point without a unique tangent line. It seems obvious that there is no continuous deformation of figure-of-eight to a circle in which all the intermediate curves remain kink-free (see Fig. 1).

Figure 2 shows another curve that clearly has a kink-free deformation to a circle.

Let us make this precise. By a (closed planar) curve we mean a continuous function $C:[0,1]\to E^2$, $C(0)=C(1)$, where E^2 is the Euclidean plane. The curve C is *regular* if the first derivative $C'(t)$ is defined and not equal to zero for all $t\in[0,1]$, and $C'(0)=C'(1)$. Let $h:[0,1]\times[0,1]\to E^2$ be a *homotopy* between curves C_0 and C_1, i.e., h is a continuous function and each $C_s:[0,1]\to E^2$ $(0\leqq s\leqq 1)$ is a curve, where we define $C_s(t)=h(s,t)$ $(t\in[0,1])$. The homotopy is *regular* if each C_s is regular. Two regular curves are *regularly equivalent* if there is a regular homotopy between them. A classical result known as the Whitney-Graustein theorem [5], [1] says that two curves are regularly equivalent if and only if they have the same winding number (up

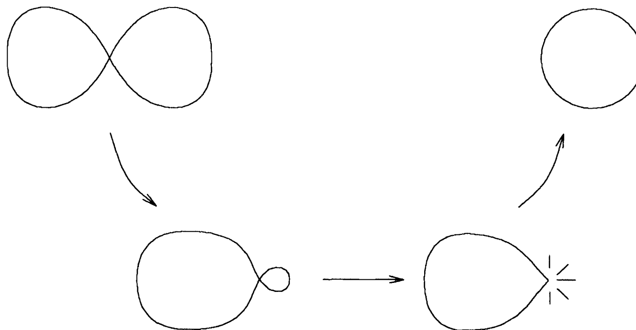


FIG. 1. *Transforming a figure-of-eight to a circle: kink appears.*

* Received by the editors February 3, 1988; accepted for publication (in revised form) September 12, 1990.

† Informatik, Universitaet des Saarlandes, D-6600 Saarbruecken, Federal Republic of Germany. The work of this author was supported by Deutsche Forschungsgemeinschaft grant Me6-1.

‡ Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012. The work of this author was supported by National Science Foundation grants DCR-84-01898 and DCR-84-01633.

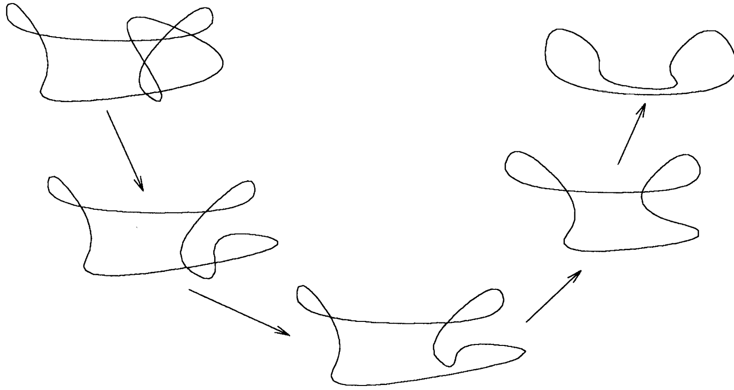


FIG. 2. *A kink-free deformation.*

to sign). Here, the winding number of a regular curve $C:[0,1]\to E^2$ may be defined as follows. The "tangent map" of C is defined to be

$$\omega_C : S^1 \to S^1,$$

where S^1 is the unit circle and for any point $\theta=\theta(t)=\exp(2\pi t\sqrt{-1})\in S^1$, $0\leqq t\leqq 1$, we let $\omega_C(\theta)=C'(t)/|C'(t)|$. Note that ω_C is well defined since C is regular. Then the *winding number* of C is

$$\frac{1}{2\pi}\int_{S^1} d\omega_C.$$

The winding number is an integer. For instance, the winding number of the figure-of-eight is zero and the winding number of the circle is ± 1 (depending on the orientation of the curve C). Since these two curves have have distinct winding numbers, the Whitney-Graustein theorem confirms our intuition that they are not regularly equivalent.

We should point out a closely related result of Hopf.[1] Hopf's theorem [2] says that two maps $f, g: S^1\to S^1$ are homotopic to each other if and only if they have the same winding number. In fact, Hopf's theorem generalizes to higher dimensions for maps on the n-sphere S^n.

The purpose of this paper is to give a constructive version of the Whitney-Graustein theorem. The rest of this paper is organized as follows. In § 2, we formulate the discrete (polygonal) version of regular curves and regular equivalence. Section 3 introduces a normal form for polygons. In § 4, we prove the Whitney-Graustein theorem for polygons. An algorithm is developed in § 5 using the insights from the proof. Section 6 concludes the paper.

2. Classification of regular polygons. Computational issues arising from the Whitney-Graustein theorem include asking for a procedure to decide equivalence of two given regular curves and to construct a regular homotopy between two equivalent curves. To obtain computational complexity results, we discretize these questions. The natural candidates for discretized regular curves would be polygons (i.e., closed polygonal paths). Unfortunately, polygons are never "regular" since they automatically

[1] In our original paper [3], we mistakenly referred to the Whitney-Graustein theorem as Hopf's theorem. We are grateful to Gert Vegter for setting the record straight.

have kinks at their vertices. This would seem to destroy any hope for a Whitney-Graustein theorem for polygons. It turns out that we can isolate the essential features of the original "regularity" assumption, and transfer these features into the polygonal setting. So there are "regular" polygons after all.

DEFINITION. A *path* Π is specified by a sequence

$$\Pi = \langle v_1, v_2, \cdots, v_n \rangle, \qquad n \geqq 2$$

of points which we call *vertices*. The *initial* and *final* vertices are v_1 and v_n, respectively, and if $n \geqq 3$, we call $v_2, \cdots, v_{n-1}$ the *interior vertices*. We require adjacent vertices to be distinct: $v_i \neq v_{i+1}$ for $i = 1, \cdots, n-1$. The edges of the path are the line segments $[v_i, v_{i+1}]$ for $i = 1, \cdots, n-1$. The *reverse* of $\langle v_1, v_2, \cdots, v_n \rangle$ is $\langle v_n, \cdots, v_2, v_1 \rangle$. A *closed* path is one of the form

$$\Pi = \langle v_1, \cdots, v_n, v_{n+1} \rangle, \qquad n \geqq 2$$

such that $v_{n+1} = v_1$. Intuitively, if we identify the first and last vertices of a closed path, then we would like to consider two closed paths as equivalent if one sequence can be obtained from the other by a cyclic shift. More precisely, two closed paths Π, Π' are *cyclically equivalent* if $\Pi = \langle v_1, \cdots, v_n, v_1 \rangle$ and Π' is equal to

$$\langle v_i, v_{i+1}, \cdots, v_n, v_1, v_2, \cdots, v_{i-1}, v_i \rangle$$

for some $i = 1, \cdots, n$. An *oriented polygon P* on $n \geqq 2$ vertices is defined as the cyclic equivalence class of some closed path $\langle v_1, \cdots, v_n, v_1 \rangle$. The *reverse* of an oriented polygon is defined as expected. A *nonoriented polygon P* is the class of closed paths cyclically equivalent to some closed path or its reverse. For short, "polygon" is understood to mean nonoriented polygon. If P is the polygon consisting of closed paths cyclically equivalent to $\langle v_1, \cdots, v_n, v_1 \rangle$ or its reverse, we denote P by

$$P = (v_1, \cdots, v_n).$$

So P could also be written as $(v_n, v_{n-1}, \cdots, v_1)$, and also $(v_2, v_3, \cdots, v_n, v_1)$, etc.

DEFINITION. Let v_i be a vertex of a path $\langle v_1, \cdots, v_n \rangle$ or a polygon $(v_1, \cdots, v_n)$, where v_i is an interior vertex in case of the path. Then v_i is a *kink* if the two edges $[v_{i-1}, v_i]$ and $[v_i, v_{i+1}]$ incident on v_i overlap. By definition, the initial and final vertices of a path are never kinks, and the vertices of a two-vertex polygon are always kinks. A path or polygon is *regular* if it has no kinks. An oriented polygon is *regular* if its nonoriented counterpart is regular.

Here, as throughout the paper, arithmetic on subscripts of vertices of a polygon P is modulo n, the number of vertices of P. Note that regularity precludes neither nonadjacent vertices from coinciding nor nonadjacent edges from overlapping.

Let v_i be an interior vertex of a regular path $\langle v_1, \cdots, v_n \rangle$. Then the *turning angle* at v_i is defined to be the angle of absolute value less than π that is equal to $\theta_i - \theta_{i-1} \pmod{2\pi}$, where θ_i is the orientation of the ray from v_i through v_{i+1}. Note that if v_i were a kink, we would have an ambiguous choice of either π or $-\pi$. We say the path makes a *right-turn, left-turn, no-turn* at v_i according as the turning angle at v_i is negative, positive, or zero, respectively. Let P be a regular oriented polygon that is cyclically equivalent to the closed path $\Pi = \langle v_1, v_2, \cdots, v_n, v_1 \rangle$. We define the *turning angle* of a vertex v_i of P as follows. If $i \neq 1$, then this is equal to the turning angle of v_i when v_i is regarded as a vertex of Π; if $i = 1$, then this is the angle of absolute value less than π that is equal to $\theta_1 - \theta_n \pmod{2\pi}$. The *winding number* of an oriented regular polygon is the sum of the turning angles at each of its vertices, divided by 2π. We see that the winding number of a regular oriented polygon is an integer which is equal to

the negative of the winding number of the reverse oriented polygon. Hence we may define the *winding number* of a nonoriented polygon to be equal to the absolute value of the winding number of any one of its two oriented versions.

It is capturing this notion of turning angle that necessitates our regularity requirement. Henceforth, we assume all polygons and paths are regular unless otherwise noted.

We introduce three types of *regular transformations* of a polygon $P=(v_1,\cdots,v_n)$. Let $i=1,\cdots,n$.

(T0) Insertion. We may transform P to

$$Q=(v_1,\cdots,v_i,u,v_{i+1},\cdots,v_n),$$

where u is a point in the relative interior of the edge $[v_i, v_{i+1}]$.

(T1) Deletion. We may transform P to

$$Q=(v_1,\cdots,v_{i-1},v_{i+1},\cdots,v_n)$$

provided v_{i-1}, v_i, v_{i+1} are collinear (and hence, by regularity of P, v_i lies strictly between v_{i-1} and v_{i+1}).

(T2) Translation. We may transform P to

$$Q=(v_1,\cdots,v_{i-1},u,v_{i+1},\cdots,v_n),$$

where u is any point such that for all $0\leqq t\leqq 1$, the polygon $Q_t=(v_1,\cdots,v_{i-1},(1-t)v_i+tu,v_{i+1},\cdots,v_n)$ is regular. In particular, $Q=Q_1$ is regular.

It is not hard to characterize the possible choices for the point u in (T2). Relative to vertex v_i, we define two *forbidden cones* (at v_{i-1} and at v_{i+1}, respectively): the forbidden cone at v_{i-1} is bounded by the two rays emanating from v_{i-1}, one ray directed towards v_{i-2} and the other directed away from v_{i+1}. See Fig. 3. Of the two complementary cones bounded by these rays, we choose the one that does not contain v_i. The forbidden cone at v_{i+1} is similarly defined, being bounded by the two rays emanating from v_{i+1} (one directed towards v_{i+2} and the other directed away from v_{i+1}). Each cone is a closed region so it includes the bounding rays. In (T2), we are free to choose any u as long as u is not in the union of the two forbidden cones.

DEFINITION. We say that two polygons P, Q are (*regularly*) *equivalent* if one can be transformed to the other by a finite sequence of regular transformations.

Clearly, insertions and deletions are inverse operations but translations are "self-inverses." It is easily seen that regular equivalence is an "equivalence" relation in the

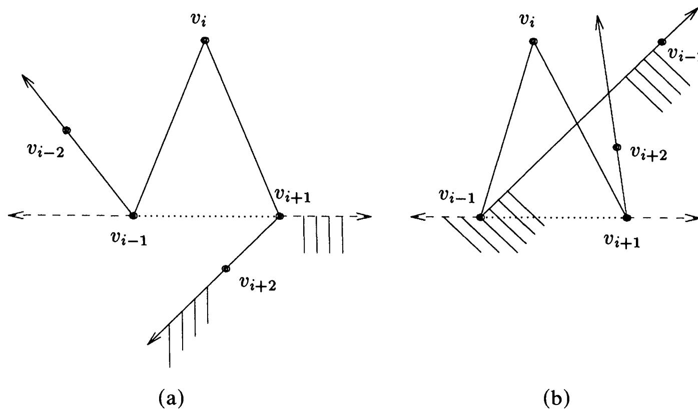


FIG. 3. (a) *Forbidden cone at* v_{i+1}. (b) *Forbidden cone at* v_{i-1}. *Forbidden cones are shaded.*

usual mathematical sense. Also, our transformations preserve winding number. We would expect the discrete analogue of the Whitney-Graustein theorem to assert that winding number is a complete invariant for regular equivalence among polygons. Towards this end, we will define a "normal form polygon" for each winding number and show that a quadratic number of regular transformation steps suffices to bring any polygon to one of these normal forms. Our algorithm finds these quadratically many steps in linear time (*sic*). (The subtitle of this paper refers to this transformation sequence from P to its normal form $\hat{P}$ as "untangling.") This is sufficient to solve the problem of finding the regular transformations from any polygon P to any other equivalent polygon Q: first transform P to its normal form $\hat{P}$ and then apply *in reverse order* the inverse of each of the transformation steps that takes Q to its normal form $\hat{Q}(=\hat{P})$.

To think about what normal forms might be desirable, we note (see Fig. 4) that the triangle and the bow-tie are obvious candidates for normal forms. Perhaps less convincingly, the 5-point star (5-star) also seems like a good candidate for a normal form.

Figure 5 illustrates a sequence of regular transformations (some steps are omitted) from the "Victoria Cross" to the 7-star polygon (with one fewer vertex). It does not seem obvious how we can systematically transform the Victoria Cross to the 7-star polygon, even if we were told that such a sequence of transformations exist.

Remark. Any smooth closed curve can be approximated by a polygon, and any homotopy between smooth curves can be discretely approximated by a series of our (T0), (T1), (T2) transformations. So in some sense, we have solved the original question of constructing regular homotopies between equivalent regular curves.

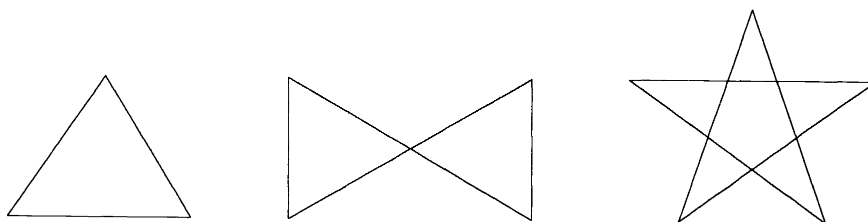


FIG. 4. *The triangle, bow-tie, and 5-star.*

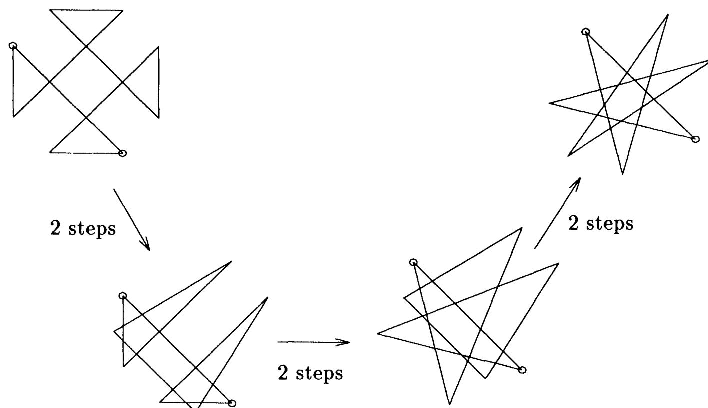


FIG. 5. *Reduction of the Victoria Cross.*

3. Star polygons.

DEFINITION. A polygon is *reducible* if it is equivalent to one with fewer vertices. It is *irreducible* otherwise.

We can easily check that the triangle, bow-tie, and 5-star cannot be transformed by (T1) or (T2) transformations into any polygon with fewer vertices. But it turns out that even allowing (T0) transformations (which insert new vertices), these polygons cannot be transformed into ones with fewer vertices; in other words, they are irreducible. Since they have different numbers of vertices, it follows that they are inequivalent to each other. In fact, any polygon with at most six vertices is equivalent to one with three candidates. In particular, there is no irreducible polygon on six vertices.

The 5-star is equivalent to the polygons in Fig. 6. The first polygon (the Fox) in Fig. 6 has the minimum number of self-intersections among its equivalence class, so one could argue that the Fox is a better choice for normal form than the 5-star.

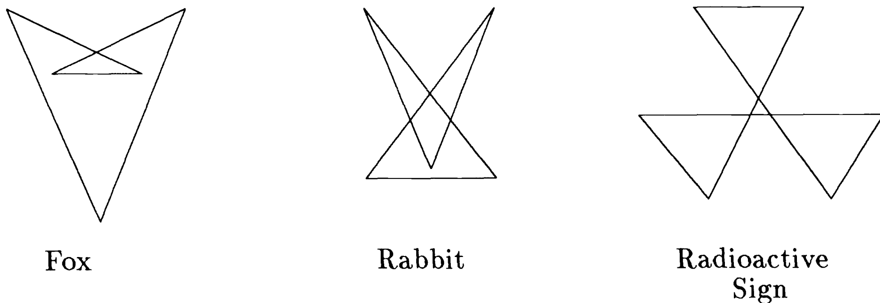


FIG. 6. *Equivalent to the 5-star: a Fox, a Rabbit, and a Radioactive Sign.*

LEMMA 1. *Every polygon P can be transformed by* (T2) *transformations into a polygon Q, all of whose vertices are distinct and lie on a circle C. Here C is any circle that contains P in its interior.*

Proof. Let C be such a circle. Recall that for each vertex v of P, we have defined two "forbidden cones." The "nonforbidden region" of v is the complement of the union of these two cones. It is not hard to see that the nonforbidden region of v contains a nonemtpy open cone K of infinite rays emanating from v. Any ray R from this cone K intersects the circle C at some point u, and a translation will take v to u. Since K is open, we can choose R to ensure that u is distinct from each vertex of P already on C. This can be repeated for successive vertices v of P. □

Henceforth we assume that all vertices of polygons and paths are distinct (with the obvious exclusion for closed paths) and lie on some circle. The circle depends on the individual polygon or path.

DEFINITION. A path $\Pi = \langle v_1, \cdots, v_n \rangle$ is called a *star path* if each edge $e_i = [v_i, v_{i+1}]$ (for $i = 1, \cdots, n-1$) intersects each of the edges

$$e_1, e_2, \cdots, e_{i-1}.$$

See Fig. 7. Since edges are closed line segments, e_i $(i \geqq 2)$ always intersects e_{i-1}. A polygon $P = (v_1, \cdots, v_n)$ is called an *n-star* if for some choice of an *initial* vertex v_i, $i = 1, \cdots, n$, the path

$$\Pi_i = \langle v_i, v_{i+1}, \cdots, v_n, v_1, n_2, \cdots, v_{i-1} \rangle \tag{1}$$

is a star path.

This terminology agrees with what we have called a 5-star. A triangle is a 3-star and a bow-tie a 4-star. Figure 8 shows the next few n-stars.

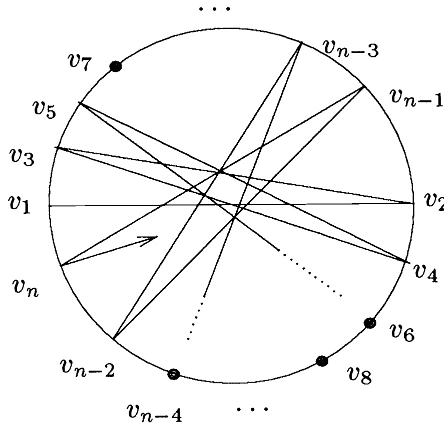


FIG. 7. *Star path.*

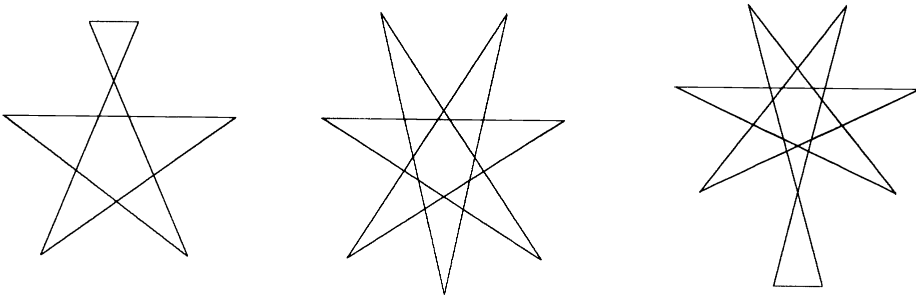


FIG. 8. 6-, 7-, *and* 8-*stars.*

Let us make some simple observations about star paths. A path is *right-turning* (respectively, *left-turning*) if it makes only right-turns (respectively, left-turns) at each of its interior vertices. A path is right-turning if and only if its reverse is left-turning. We see that a star path of length $n \geqq 3$ is either left-turning or right-turning. Let $\Pi_1 = \langle v_1, \cdots, v_n \rangle$ be a star path. For each $i = 1, \cdots, n$, let Π_i be the path in (1). Π_i is basically some cyclic shift of Π_1. If n is odd, then Π_i is a star path for each i. For even values of $n \neq 4$, Π_i is not a star path for all $i > 1$. For $n = 4$, Π_1 and Π_3 are the only star paths.

Notation. If $u_1, \cdots, u_k$, $k \geqq 3$, are distinct vertices of polygon P lying on some circle C, we write

$$u_1 < u_2 < \cdots < u_k \tag{2}$$

to mean that, as we traverse the circle C in a clockwise fashion starting from u_1, we will meet the vertices $u_1, u_2, \cdots, u_k$ in this order (though other vertices not among the u's may intervene). Thus $u_1 < u_2 < \cdots < u_k$ is equivalent to $u_2 < u_3 < \cdots < u_k < u_1$, etc. We call a list of the form (2) a *cyclic permutation* on the vertices $u_1, \cdots, u_k$. So any polygon or path whose vertices include $u_1, \cdots, u_k$ *induces* a cyclic permutation on $u_1, \cdots, u_k$. Note that the cyclic permutation induced by a path or its reverse are the same.

For example, $\Pi = \langle v_1, \cdots, v_n \rangle$ is a left-turning star path if and only if Π induces the following cyclic permutation on the points $v_1, \cdots, v_n$:

$$v_1 < v_3 < v_5 < \cdots < v_{\text{odd}} < v_2 < v_4 < v_6 < \cdots < v_{\text{even}} \tag{3}$$

where $v_{\text{odd}} = v_n$, $v_{\text{even}} = v_{n-1}$ if n is odd, and $v_{\text{odd}} = v_{n-1}$, $v_{\text{even}} = v_n$ if n is even. Figure 7 illustrates the case where n is even.

The following gives a method for moving a vertex to another given position.

LEMMA 2 (moving a vertex). *Let $P = (u_1, \cdots, u_n)$ be a polygon whose vertices lie on a circle C. Let x be any point of C. Let $A \subseteq C$ be one of the two closed arcs bounded by u_1 and x. Call an edge of P <u>active</u> if at least one of its endpoints is in A. The vertices in A are thus partitioned into <u>connected components</u> where two vertices are "connected" if they are joined by a path of active edges. Let U be a union of any number of connected components with the provision that $u_1 \in U$ and (if x is a vertex of P) $x \in U$. Then for any open neighborhood N of x, there is a sequence of $|U|$ translations that moves u_1 to x, moves the vertices in $U - \{u_1\}$ into $(C \cap N) - A$, and keeps all the remaining vertices of P fixed. Moreover the relative ordering of the elements in U is preserved.*

Proof. The vertices in U can be ordered according to increasing distance from x, where distance is measured along the arc A. See Fig. 9. Starting from the vertex in U that is closest to x, we translate each one in turn into $(C \cap N) - A$. The last vertex to be translated would be u_1 itself, and this can be translated to x. To see why this works, note that a vertex u_j can be translated to any position in C provided the induced cyclic permutation on $u_{j-2}, u_{j-1}, u_j, u_{j+1}, u_{j+2}$ is preserved. So when we try to translate $u_j \in U$ as described, we see that each of $u_{j-2}, u_{j-1}, u_{j+1}, u_{j+2}$ either has been moved into $(C \cap N) - A$ already or lies outside of A. In any case, we can translate u_j to some position in $(C \cap N) - A$ which preserves the induced permutation of U. $\square$

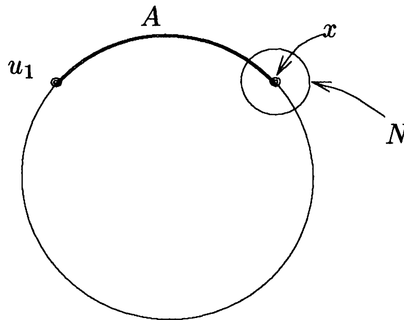


FIG. 9. *Moving clockwise from u_1 to x.*

The transformation described by this lemma may be described as "moving u_1 to x relative to (A, U, N)." The smallest choice for U is the union of the connected component of u_1 with the connected component of x (regard the latter component to be empty if x is not a vertex). The largest choice for U is the set of all the vertices in A. If N is chosen small enough so that N contains no vertices of P, then we refer to the move corresponding to the smallest choice for U as a *weak move clockwise/counterclockwise from u_1 to x* (where A is the arc clockwise/counterclockwise from u_1 to x). If U is the largest possible choice, it is similarly known as a *strong move.* Note that a strong move preserves the induced cyclic permutation of P.

LEMMA 3. *Let $P = (u_1, \cdots, u_n)$ and $A = (v_1, \cdots, v_n)$ be two polygons with all their vertices lying on a common circle C. Let σ and τ be the cyclic permutations on the u's*

and v's induced by P and Q, respectively. If σ and τ are similar (in the sense that after renaming each u_i by v_i, they are identical), then P and Q are regularly equivalent in less than or equal to $1.5n$ *steps.*

Proof. As usual, the u's are pairwise distinct and so are the v's; without loss of generality, assume σ and τ are both the identity. Using the above lemma, we make a strong move clockwise or counterclockwise from u_1 to v_1; of the two possible clock directions, we could choose the one using at most $n/2$ translational steps. Applying the lemma again, we make a strong move from either u_2 to v_2 or v_2 to u_2 using only one translational step. In general, assuming $u_1, \cdots, u_{i-1}$ is coincident with $v_1, \cdots, v_{i-1}$, we can move either u_i to v_i or v_i to u_i in only one step. □

For instance, any n-star $(v_1, \cdots, v_n)$ induces the cyclic permutation (3). Hence this lemma tells us that any two n-stars (which we may assume to lie on a common circle C) are regularly equivalent. In view of this, we henceforth refer to "the n-stars" as if these were unique for each n.

LEMMA 4. *Let n, m be odd positive integers or equal to four. If $n \neq m$, then the n-star and the m-star are inequivalent.*

Proof. We check that the winding number of the 4-star is zero and for each positive integer k, the $(2k+1)$-star has winding number k. The result then follows from the fact that the winding number of a polygon is unchanged by any regular transformation. □

This lemma supplies us with an infinite list of inequivalent polygons. We will prove that every regular equivalence class is represented in this list.

4. The Whitney–Graustein theorem for polygons. The main result of this section is Theorem 5.

THEOREM 5 (canonical form). *Every polygon can be transformed by a sequence of* (T1) *and* (T2) *transformations into an n-star, for some n that is either odd or equal to four.*

COROLLARY 6. *An n-star is irreducible if and only if $n = 4$ or n is odd.*

Proof. Suppose that an n-star is irreducible. Then the theorem implies that n must be four or odd. Conversely, let $n = 4$ or odd. If an n-star were reducible, then the theorem shows that it would be reducible to an m-star for some $m < n$ where $m = 4$ or odd. This contradicts the previous lemma that the n- and m-stars are inequivalent. □

We prove the canonical form theorem by a sequence of lemmas.

A polygon that can (respectively, cannot) be transformed to one with fewer vertices using just (T1) and (T2) transformations will be called *semireducible* (respectively, *semi-irreducible*). (Of course, in view of Theorem 5, semireduciblity turns out to be the same concept as reducibility.)

Notation. For compactness, we will usually write only the *indices* (i.e., subscripts) of vertices in place of the vertices themselves. Thus we write $P = (1, 2, \cdots, n)$ for a polygon on n vertices. Combined with an earlier notation, we may write "$1 < 3 < 2$" to mean "$v_1 < v_3 < v_2$": this is hopefully not too confusing.

The following simple fact is often used.

LEMMA 7 (deleting a vertex). *Suppose $P = (1, \cdots, n)$ $(n \geqq 5)$ is such that the pair of edges* $[1, 2]$ *and* $[3, 4]$ *does not intersect, and also the pair* $[2, 3]$ *and* $[4, 5]$ *does not intersect. Then P is equivalent to* $(1, 2, 4, 5, \cdots, n)$ *after a* (T2) *followed by a* (T1) *transformation. In other words, we may delete index* 3.

Proof. See Fig. 10. The nonintersection assumptions of the lemma imply that the interior of the triangle $\Delta 234$ is nonforbidden for vertex 3. Hence we can translate vertex

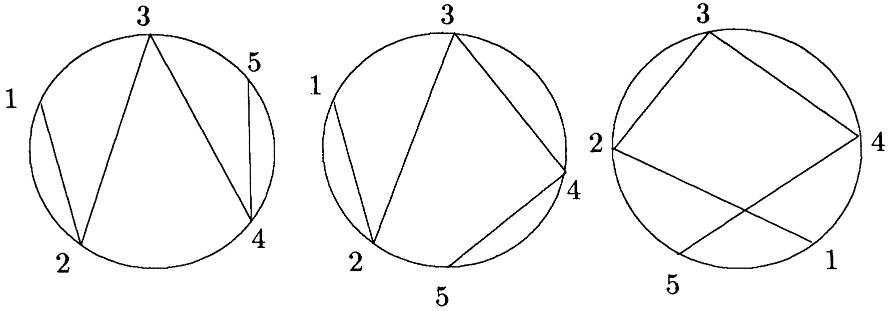


FIG. 10. *Can delete index* 3.

3 to the midpoint of edge [2, 4] by a (T2) transformation. Next, a (T1) transformation eliminates vertex 3. □

Henceforth, whenever we delete vertices, it is by appeal (usually implicit) to this lemma.

We say that $P=(1,2,\cdots,n)$ *contains an N-shape* if $n\geqq 4$ and for some choice of index i, we have

$$i<i+1<i+3<i+2$$

(Fig. 11a) or

$$i<i+2<i+3<i+1$$

(Fig. 11b). We call $(i,\ i+1,\ i+2,\ i+3)$ an N-*shape.*

LEMMA 8. *A semi-irreducible polygon* $P=(1,2,\cdots,n)$ *does not contain an N-shape unless* $n=4$.

Proof. By way of contradiction, assume that P has an N-shape. By symmetry, assume that $1<2<4<3$ (Fig. 12).

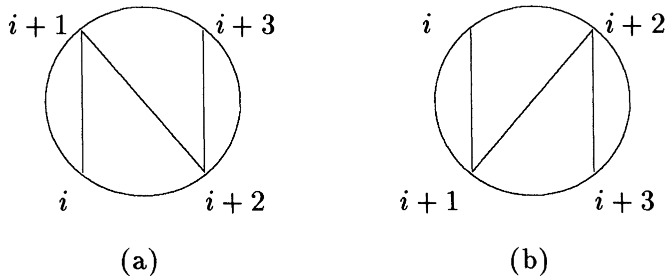


FIG. 11. *An* N-*shape.*

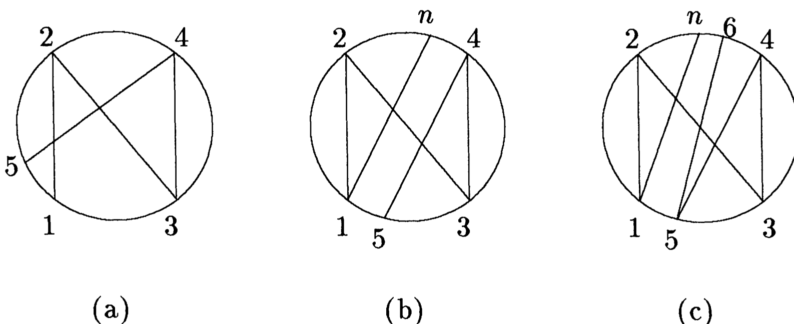


FIG 12. *Reduction of an* N-*shape.*

The result is true for $n=4$, so suppose $n \geqq 5$. Since P is semi-irreducible, by the previous lemma, the edge $[4, 5]$ must intersect $[2, 3]$; hence $3<5<2$. Similarly, $2<n<3$. This shows that $n \neq 5$ so assume $n \geqq 6$. If $1<5<2$ (Fig. 12(a)) then we can translate index 2 so that $1<2<5$ (this translation can occur because $2<n<3$). Then we can delete 3, a contradiction. Therefore, we have $3<5<1$. By symmetry, we have $2<n<4$. The situation is shown in Fig. 12(b).

If $n=6$ then it is easy to see that P is semireducible. Otherwise, consider the location of index 6. There are two cases. First suppose $n<6<4$ (Fig. 12(c)). If index 7 is such that $5<7<1$ then we can delete index 5, a contradiction. Otherwise we may translate index 5 so that $1<5<2$, which reduces to a previous case (Fig. 12(a)). In the second case, $6<n<4$ and we can translate index 4 so that $4<n<3$ (and hence $4<n<2$). This again reduces to a previous case. □

Note that a polygon does not contain any N-shape if and only if its oriented versions are left-turning or right-turning.

COROLLARY 9. *An n-star is semireducible if n is even and not equal to four.*

Proof. If $\langle 1, \cdots, n\rangle$ is a star path and n is even, then $(n, 1, 2, 3)$ forms an N-shape. Since $n \neq 4$, the previous lemma implies that $(1, \cdots, n)$ is semireducible. □

We say that $P=(1, 2, \cdots, n)$ *contains a* U-*shape* if $n \geqq 4$ and for some choice of index i, we have

$$i<i+1<i+2<i+3$$

(Fig. 13(a)) or

$$i<i+3<i+2<i+1$$

(Fig. 13(b)). We call $(i, i+1, i+2, i+3)$ a U-*shape.*

LEMMA 10. *A semi-irreducible polygon* $P=(1, 2, \cdots, n)$ *cannot contain a* U-*shape.*

Proof. We can easily check this for $n=3$, 4, and 5; so assume that $n \geqq 6$. Suppose indices (2, 3, 4, 5) form a U-shape as in Fig. 14, $2<5<4<3$.

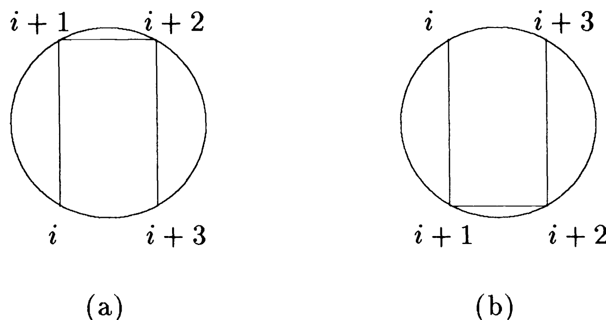


FIG. 13. *A* U-*shape.*

Since index 3 cannot be deleted, we must have $4<1<3$; similarly, since index 4 cannot be deleted, $4<6<3$. Suppose that the relative positions of indices 1 and 6 satisfy

$$4<1<6<3, \tag{4}$$

as in Fig. 14(a). Then we may translate index 3 so that $1<3<6$ and then delete index 4, a contradiction. Hence we may assume the situation of Fig. 14(b), with $4<6<1<3$.

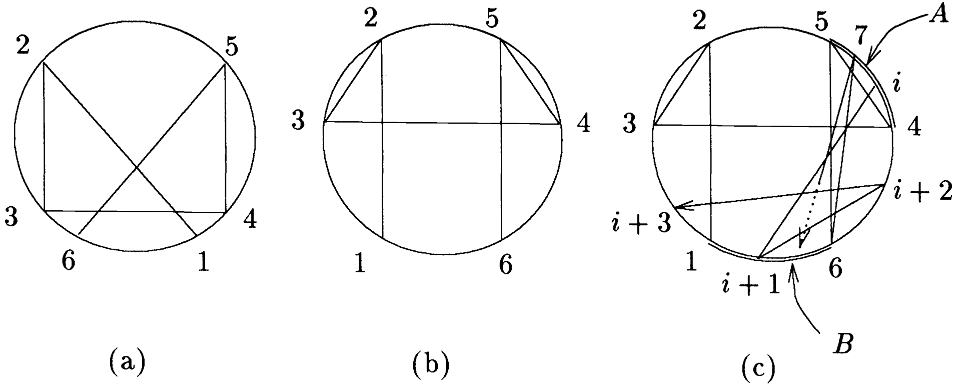


FIG 14. *The elimination of* U-*shape.*

Note that the path $\langle 1, \cdots, n, 1\rangle$ is left-turning since P contains no N-shape. This means $i < i-1 < i+1$ holds for all i. In particular, $1 < n < 2$ and $5 < 7 < 6$. This implies $n \geqq 8$.

Let A be the arc clockwise from index 5 to index 4, including the index 5 but not 4. Similarly, B is clockwise from index 6 to index 1, including 6 but not 1. First note that we can translate index 7 into A. If index 8 is not in B then we can translate index 6 so that (4) holds, a contradiction. So index 8 lies in B and $n \geqq 9$. But $7 < 9 < 8$ implies $n \geqq 10$.

Suppose inductively that we have shown for some odd $i \geqq 7$ (a) the indices $5 < 7 < \cdots < i-2 < i$ all lie in A, (b) the indices $6 < 8 < \cdots < i-1 < i+1$ all lie in B, and (c) $n \geqq i+3$ (see Fig. 14(c)).

We claim that either we can extend the inductive assumptions (a)-(c) or we get a contradiction. First we can translate $i+2$ into A so as to extend (a). Next, if $i+3$ does not lie in B then let x be a point such that $1 < x < 3$ and $1 < x < i+3$. We make a weak move from index 6 clockwise to x (see Lemma 2). Note that neither of the edges $[3, 4]$ and $[i+2, i+3]$ is active relative to the arc clockwise from 6 to x, and hence index 1 is fixed by the weak move. But now we are in the situation of (4) again, a contradiction. So assume that $i+3$ is in B. It is easy to see from this that we have extended assumption (b). Now $i+3$ is in B implies $n \geqq i+4$. But since $i+2 < i+4 < 1$, we see that in fact $n \geqq i+5$, which is assumption (c) extended.

Since we cannot extend the inductive assumptions indefinitely, we will eventually derive a contradiction. □

We give one more lemma before proving the main result of this section.

LEMMA 11. *Let* $P = (1, 2, \cdots, n)$, $n \geqq 5$, *be any semi-irreducible polygon. Then* $\Pi = \langle 1, 2, \cdots, n\rangle$ *is a star path.*

Proof. We now know that P contains no N- and no U-shapes. We will show that if $\Pi_v = \langle 1, 2, \cdots, v\rangle$ (for $v = 3, \cdots, n-1$) is a star path, then Π_{v+1} is a star path. Consider the situation in Fig. 15 (without loss of generality, assume $1 < 3 < 2$).

The induction basis $v = 3$ follows from the previous two lemmas since if Π_4 is not a star path, then it forms a U- or an N-shape. The same remark holds for $v = 4$, so let $v \geqq 5$.

First suppose v is odd. If Π_{v+1} is not a star path then $1 < v+1 < v-2$. Let x be a point such that $1 < x < 3$ and $1 < x < v+1$. We do a weak move clockwise from 4 to x. Note the edges $[2, 3]$ and $[v, v+1]$ are not active. Hence index 1 is left fixed by the weak move. So $(1, 2, 3, 4)$ is now a U-shape, a contradiction.

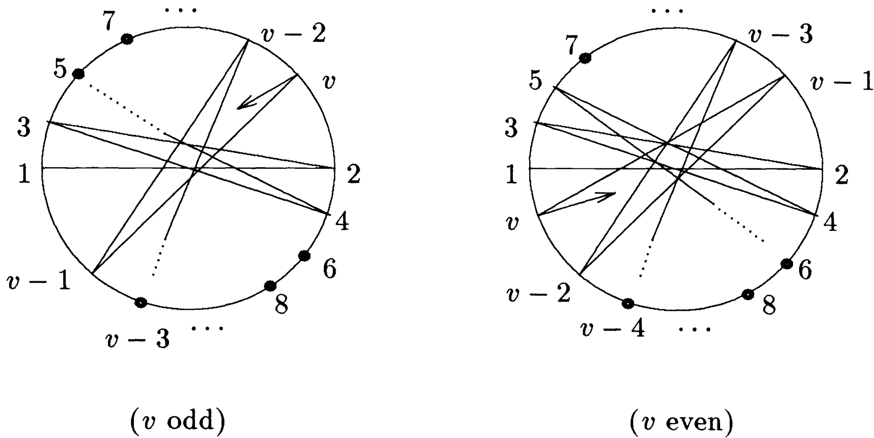


FIG. 15. *Star paths from semireducible polygons.*

Suppose v is even. If Π_{v+1} is not a star path then $2<v+1<v-2$. Let x be such that $2<x<v+1$ and A be the arc from $v-2$ counterclockwise to x. We do a weak move from $v-2$ to x. Note that edges $[1, 2]$ and $[v-1, v]$ are not active. Hence $v+1$ is left fixed. So $(v-2, v-1, v, v+1)$ is now a U-shape, a contradiction. □

The main result, Theorem 5, now follows: given any polygon P, we can reduce it by (T1) and (T2) transformations until it is semi-irreducible. Then, by the last lemma, the result must be an n-star. By Corollary 9, n is odd or equal to four. This proves the theorem.

Since there is a unique canonical form for each winding number and conversely, this proves that the winding number is a complete invariant for regular equivalence. This is the polygonal version of the Whitney-Graustein theorem.

5. Algorithm. The proof of the canonical form theorem contains an implicit quadratic-time algorithm to transform a polygon to its normal form. We now give a *real-time* algorithm to construct a similar sequence of quadratic transformation steps—this apparent paradox is soon clarified.

The algorithm *processes* the input vertices in order. In the generic situation, the vertices $v_1, \cdots, v_{j-1}$ have been processed and the polygon has been transformed into an equivalent polygon

$$P_j=(u_1, \cdots, u_{i-1}, v_j, \cdots, v_n).$$

Furthermore, we assume that

$$\langle u_1, \cdots, u_{i-1}\rangle$$

forms a star path. (Throughout this section, the indices i and j will have this special meaning.) The *current vertex* being processed is v_j although our algorithm may look ahead slightly, up to v_{j+3}. (The algorithm halts after "wrapping around" to process $v_{n+2}=v_2$.)

An interesting feature of the algorithm is that it uses only $O(1)$ runtime memory. More precisely, at the moment of processing v_j, we only need in the active memory (i) the values of the indices i, j; (ii) the values of the vertices

$$v_j, v_{j+1}, v_{j+2}, v_{j+3};$$

and (iii) the *sign information*: whether the paths

$$\langle u_{i-2}, u_{i-1}, v_j\rangle \quad \text{and} \quad \langle u_{i-1}, v_j, v_{j+1}\rangle$$

make right or left turns at their respective interior vertices. The transformed vertices $u_1, \cdots, u_{i-1}$ are output in an *even stack* and *odd stack*, storing the odd and even indices, respectively.

Our algorithm runs in real time in the sense that each vertex takes $O(1)$ time to process. However, we may output for vertex v_j a sequence of $O(j)$ transformation steps. To understand how $O(j)$ steps can be described in $O(1)$ time, recall the "weak move" subroutine from Lemma 2. It turns out that whenever we output an unbounded number of transformation steps, it is through making one or two such weak moves. The vertices to be translated in such a weak move turn out to form a contiguous block of elements in the odd or even stack, and so the weak move has a constant size description (of a suitable sort). After processing all n vertices, we produce $O(n^2)$ transformation steps overall. Thus P can be transformed to an irreducible star polygon in $O(n^2)$ transformation steps.

For simplicity, we will not explicitly specify the (T1), (T2) transformation steps to output. But each step of the algorithm will be given justification and the reader can easily deduce the transformation steps needed.

Since $u_1, \cdots, u_{i-1}$ are obtained by transformations of $v_1, \cdots, v_{j-1}$, and since we may delete but never insert vertices, the relation

$$i \leqq j$$

always holds. Therefore it is unambiguous to refer to the vertices by their indices: an index k refers to v_k if $k < i$ and refers to u_k if $k \geqq j$. We assume that $j \geqq 4$. To initialize, we may let $u_i = v_i$ for $i = 1, 2, 3$ and $j = 4$. Without loss of generality, assume that the path $\langle 1, 2, \cdots, i-1\rangle$ is left-turning.

There are four cases to consider while processing vertex v_j: the triple $(i-2, i-1, j)$ (i.e., (u_{i-2}, u_{i-1}, v_j)) represent either a left turn or a right turn, and i is either odd or even. First assume that i is odd (see Fig. A).

Case A. $(i-2, i-1, j)$ is a right turn.

Case A1. $(i-1, j, j+1)$ is a left turn. We pop the even stack which contains index $i-1$. In effect, we have decremented i by one.

Justification. See Fig. B. We may delete index $i-1$. To reconstruct the sign information, note that the turns $(i-3, i-2, j)$ and $(i-2, j, j+1)$ are both left turns. (As illustrated here, we could always reconstruct such sign information without

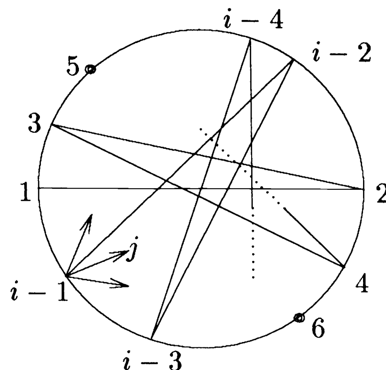


FIG. A. *Case* $i = odd$.

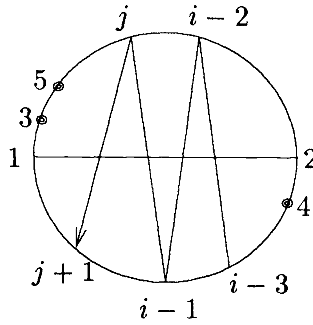


FIG. B. *Case* A1.

reexamining the vertices $i-1$, $i-2$, or $i-3$. Henceforth we will not explicitly mention this.)

Case A2. $(i-1, j, j+1)$ is a right turn. We have three subcases depending on the relative positions of indices j and $j+1$.

Case A21. $j < j+1 < 2$. After at most one translation, we may delete index $i-1$. In effect we decrement i by one.

Justification. See Fig. C. By translating index $i-2$ to some neighborhood of index 2 if necessary, we may assume that $j < j+1 < i-2$. Now delete index $i-1$.

Case A22. $j < 2 < j+1 < 1$. After $O(j)$ translations, we may delete index $i-2$. Effectively we decrement i by one.

Justification. See Fig. D. By translating index $i-2$ to some neighborhood of index 2, and by making a weak move (Lemma 2) counterclockwise from index $i-4$ to some neighborhood of index 1, we may assume that $i-4 < j < i-2$. As noted, this weak move may involve $O(j)$ translational steps but this has a constant size description. Then we are in the situation of Fig. D(a). We may now translate index $i-1$ clockwise until $i-4 < i-1 < j$ (Fig. D(b)), and then delete $i-2$. Now $\langle 1, \cdots, i-3, i-1 \rangle$ is a star path.

Case A23. $j < 1 < 2 < j+1$ (see Fig. E). We consider two subcases for index $j+2$.

Case A231. $j < 1 \leqq j+2$. By translating index j clockwise until $1 < j < 2$, we reduce this to Case A22.

Case A232. $j < j+2 < 1$. See Fig. F. Next consider subcases depending on index $j+3$.

Case A2321. $(j+1, j+2, j+3)$ is a right turn. Increment j by two.

Justification. We may delete $j+1$ and j (in that order).

Case A2322. $(j+1, j+2, j+3)$ is a left turn. See Fig. G(a). After some translations, we replace index $i-1$ by $j+2$; effectively we increment j by three.

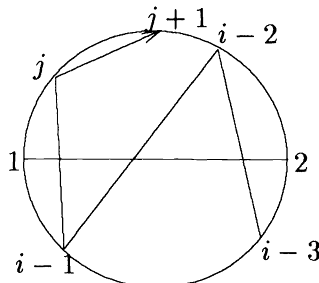


FIG. C. *Case* A21.

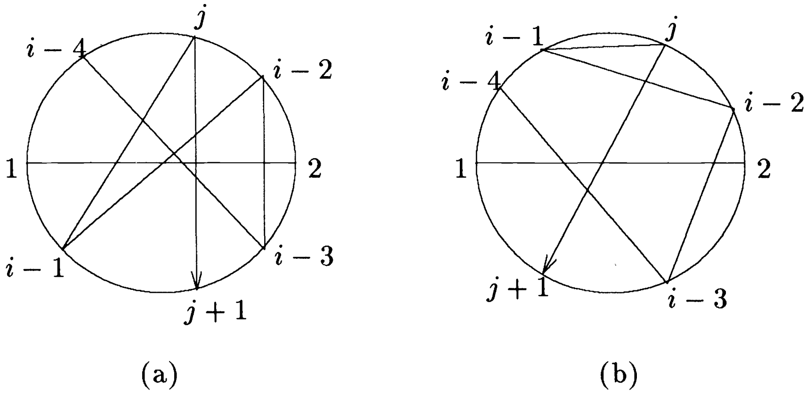


FIG. D. *Case* A22.

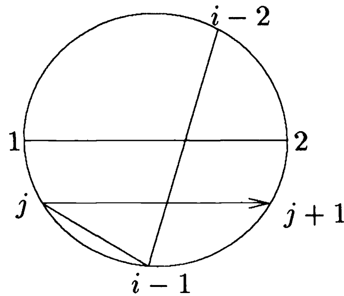


FIG. E. *Case* A23.

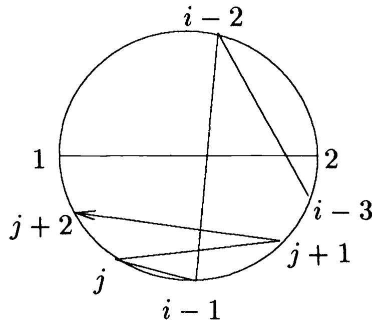


FIG. F. *Case* A232.

Justification. We translate $j+1$ counterclockwise until $1<j+1<i-2$, as in Fig. G(b). Now we may delete $i-1$ and j (in either order), and then delete $j+1$. Then $\langle 1, \cdots, i-2, j+2\rangle$ is a star path.

Case B. $(i-2, i-1, j)$ is a left turn. We have two subcases depending on whether index j lies in the arc from index $i-2$ clockwise to index 2, or from 2 clockwise to $i-1$.

Case B1. $j<2<i-2$. Push index j on the odd stack; thus we increment i and j by 1 each. See Fig. H.

Justification. $\langle 1, \cdots, i-1, j\rangle$ is a star path.

Case B2. $j<i-1<2$. We consider subcases depending on $j+1$.

Case B21. $(i-1, j, j+1)$ is a right turn. We replace index $i-1$ by index j, and increment j by one.

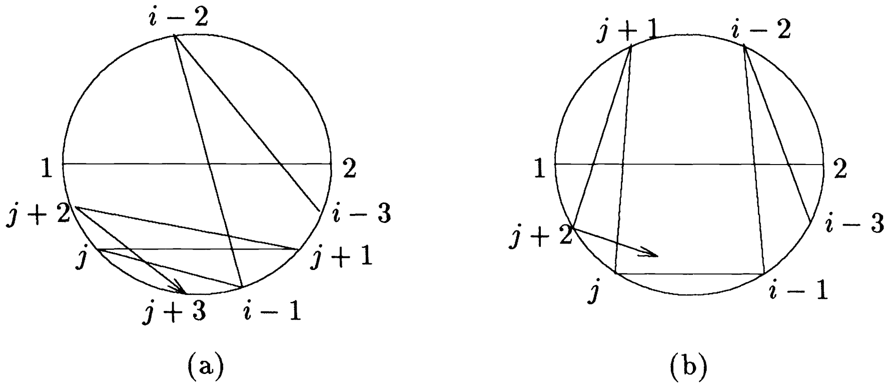


FIG. G. *Case* A2322.

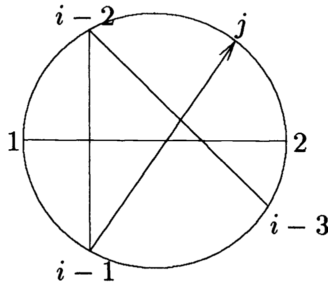


FIG. H. *Case* B1.

Justification. By a weak move counterclockwise from $i-3$ to some neighborhood of 2, we may assume that $2<i-3<j$. See Fig. J. After deleting $i-1$, $\langle 1, \cdots, i-2, j\rangle$ is a star path.

Case B22. $(i-1, j, j+1)$ is a left turn. We consider two possible positions for index $j+1$.

Case B221. $j<j+1<1$. See Fig. K. We consider the position of index $j+2$ next.

Case B2211. $j<j+2<2$. After translating index j, we can push index j onto the odd stack and increment both i and j by one.

Justification. We can translate j counterclockwise until $i-2<j<2$. Now $\langle 1, \cdots, i-1, j\rangle$ is a star path.

Case B2212. $2<j+2<j$. See Fig. L. We must examine index $j+3$ next.

Case B22121. $(j+1, j+2, j+3)$ is a left turn. Increment j by two.

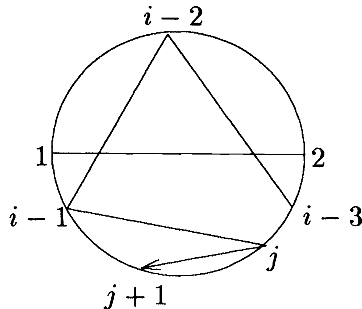


FIG. J. *Case* B21.

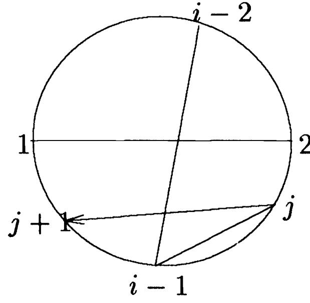


FIG. K. *Case* B221.

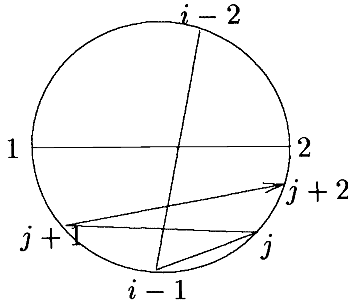


FIG. L. *Case* B2212.

Justification. We delete $j+1$ and then j.

Case B22122. $(j+1, j+2, j+3)$ is a right turn. After some translations, we may replace $i-1$ with j in the even stack, and push $j+1$ onto the odd stack. Effectively, we increment i by one and j by two.

Justification. See Fig. M(a). After making a weak move counterclockwise from index $i-3$ to some neighborhood of index 2, we may assume that $i-3<j<i-1$. Then we can translate index $j+1$ clockwise until $i-2<j+1<2$ and delete $i-1$. The sequence $\langle 1, \cdots, i-2, j, j+1\rangle$ is a star path (Fig. M(b)).

Case B222. $1<j+1<j$. After some translations, we can replace index $i-1$ by j on the stack; effectively we increment j by one.

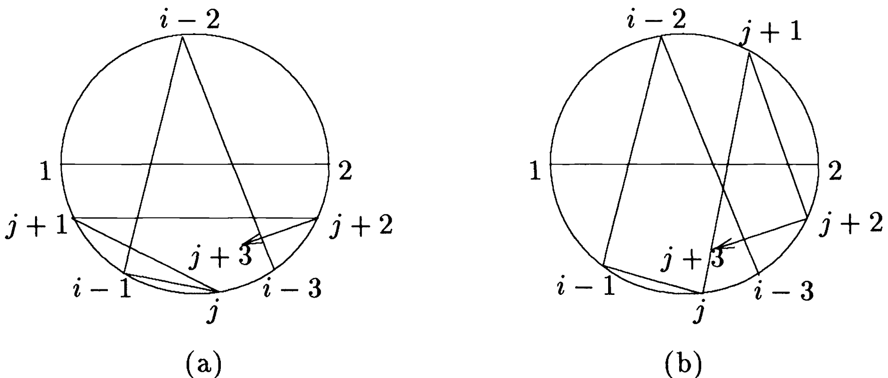


FIG. M. *Case* B22122.

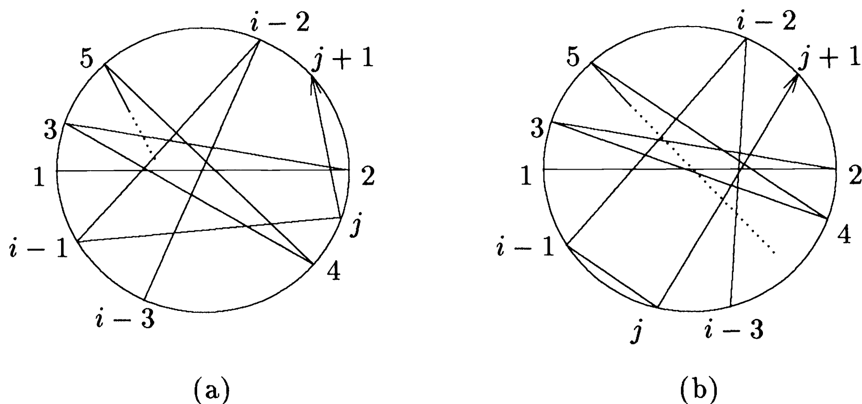


FIG. N. *Case* B222.

Justification. See Fig. N(a). We make a weak move counterclockwise from $i-2$ to some small enough neighborhood of index 1 so that $i-2<j+1<j$. Next, make a weak move counterclockwise from $i-3$ to some small enough neighborhood of index 2 so that $i-3<j<i-1$. See Fig. N(b). Now delete index $i-1$ and see that $\langle 1, \cdots, i-2, j\rangle$ is a star path.

This completes subcase B and hence the case where i is odd.

The case where i is even is similar and is left to the reader. Finally, when j reaches vertex $v_{n+2}=v_2$ again, we are done.

6. Conclusion. We have given a constructive analogue of the Whitney-Graustein theorem, resulting in a real-time algorithm to "untangle" any polygon. We emphasize that the true contribution of this work is the construction of the transformation steps: checking if two polygons are equivalent is in itself a trivial process of keeping a cumulative sum of the angles turned.

Our proof shows incidentally: (1) It suffices to use (T1), (T2) transformations to make a polygon irreducible, and (2) any two equivalent irreducible polygons are inter-transformable using only (T2) transformations.

Since the publication of these results, Vegter [4] has improved them by defining the *isothetic normal forms* for polygons, and showing that a linear number of regular transformation steps suffices to convert any polygon into its isothetic normal form.

REFERENCES

[1] M. BERGER AND B. GOSTIAUX, *Differential Geometry: Manifolds, Curves and Surfaces*, Graduate Texts in Mathematics, Vol. 115, Springer-Verlag, Berlin, New York, 1988.

[2] H. HOPF, *Abbildungsklassen n-dimensionaler mannigfaltigkeiten*, Math. Annalen, 96 (1927), pp. 209-224.

[3] K. MEHLHORN AND C. K. YAP, *Constructive Hopf's theorem: Or how to untangle closed planar curves*, in Proc. Internat. Conference on Automata, Languages and Programming, 1988, Finland, Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, New York, 1988, pp. 410-423.

[4] G. VEGTER, *Kink-free deformations of polygons*, ACM Symposium on Computational Geometry, 5 (1989), pp. 61-68.

[5] H. WHITNEY, *On regular closed curves in the plane*, Compos. Math., 4 (1937), pp. 276-284.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 622-638, August 1991

© 1991 Society for Industrial and Applied Mathematics
002

REVERSAL COMPLEXITY*

JIAN-ER CHEN† AND CHEE-KENG YAP†

Abstract. The importance of reversal complexity as a basic computational resource has only been recognized in recent years. It is intimately connected to parallel time complexity and circuit depth. In this paper, some basic techniques necessary for establishing analogues of well-known theorems on space and time complexity are developed. The main results are, for reversal-constructible functions $s(n) \geqq \log n$,

$$DSPACE(s(n)) \subseteq DREVERSAL(s(n)),$$

and a tape reduction theorem. As applications of the tape reduction theorem, a hierarchy theorem is proved and the existence of complete languages for reversal complexity is shown.

Key words. reversal, space, tape reduction theorem, reversal hierarchies, complete languages, parallel complexity

AMS(MOS) subject classifications. 68Q05, 68Q10, 68Q15

1. Introduction. The number of reversals made by tape heads during a Turing machine computation has assumed new importance as a complexity measure in complexity theory. This is because reversal complexity is intimately connected with uniform circuit depth and parallel time. For example, Hong [7] has shown that reversal in sequential machine models (including the standard Turing machines) corresponds to parallel time in parallel machine models. Also, Pippenger [9] shows that simultaneous time and reversal in Turing machines are polynomially related to simultaneous size and depth in uniform circuits. However, reversal complexity has some unexpected properties which, until recently, made researchers treat it as a curiosity rather than as a fundamental computational resource. Baker and Book [1] have shown the surprising fact that every recursively enumerable set can be recognized by a nondeterministic Turing machine making at most two tape reversals. Moreover, unlike time complexity and space complexity, which have nice properties such as "linear speedup" and "tape reduction," reversal complexity has defied attempts at finding similar theorems in the multitape Turing machine model.

Hartmanis [5] was the first to study reversal complexity. He considered on-line one-tape Turing machines and showed that for such machines, reversal and space are polynomially related. Moreover, he pointed out that for slow-growing functions, the "linear speedup theorem" for reversal complexity does not hold. Fisher [4] gave the "linear speedup theorem" for reversals in off-line one-tape Turing machines using the idea of "crossing sequences." However, the technique of Fisher does not generalize to multitape Turing machine models.

By a "tape reduction theorem for reversal complexity" we mean a result showing that, for some slow-growing function $f(n)$, and for some constant $k \geqq 1$, any multitape Turing machine using $r(n)$ reversals can be simulated by a k-tape Turing machine using $f(r(n))$ reversals. Call this an $(f(n), k)$-*tape reduction theorem.* For instance, the famous Hennie–Stearns [6] simulation achieves an $(n \log n, 2)$-tape reduction theorem for time complexity. There is also an $(n, 1)$-tape reduction result for space complexity. However, the tape reduction theorem for reversal complexity seems to have withstood

* Received by the editors June 30, 1986; accepted for publication (in revised form) October 11, 1990. This work was supported in part by National Science Foundation grant DCR-84-01898.

† Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012.

previous attempts, and the only published indication of this is a paper by Kameda and Vollmar [8]. In that paper, the authors showed that any $r(n)$ reversal-bounded k-tape Turing machine *without stationary moves* can be simulated by a 2-tape Turing machine making at most $6r(n)$ reversals. Unfortunately, it appears that such machines are too restrictive from the viewpoint of reversal complexity; in any case, their techniques do not generalize. One of the main results in this paper will be an $(n^2, 2)$-tape reduction theorem for reversal complexity, applicable to unrestricted multitape Turing machines.

We now give some definitions. Our machine model is the *multitape Turing machine.* We follow Yap [13] in the following conventions: by a "k-tape Turing machine," we mean a Turing machine with a 2-way read-only input tape and k work tapes.

$DSPACE(s)$ = the class of languages accepted by deterministic Turing machines that use at most $s(n)$ tape squares of the work tapes on inputs of length n.

$NSPACE(s)$ = the class of languages accepted by nondeterministic Turing machines that use at most $s(n)$ tape squares of the work tapes on inputs of length n.

Remark. When we say a machine M "accepts" L in space $s(n)$, we mean that for all inputs x of sufficiently large length, if x is in L, then *some* path accepts after using at most $s(|x|)$ space; but if x is not in L, then no bounds are imposed on the space used on any path. This convention is used for the other resources as well.

$DREVERSAL_k(r)$ = the class of languages accepted by k-tape deterministic Turing machines which makes at most $O(r(n))$ tape reversals on inputs of length n. It should be observed that reversals made on the input tape are counted.

Remark. The crucial point to note in this definition is that we use "$O(r(n))$" rather than "$r(n)$." This is because we do not have a linear speedup theorem on multitape Turing machines for reversal complexity.

$DREVERSAL(r) = \bigcup_k DREVERSAL_k(r)$.

A function $f(n)$ is said to be *$g(n)$-reversal-constructible* if there is a 2-tape Turing machine that, given an integer $n > 0$ in unary form, produces the unary form of $f(n)$ on one of its work tapes making at most $O(g(n))$ tape reversals. If $f(n)$ is $f(n)$-reversal-constructible, we say $f(n)$ is reversal-constructible.

In § 2, we present useful techniques for efficiently using tape reversals on Turing machines; in § 3, we show one of our main results, namely, $DSPACE(s(n)) \subseteq DREVERSAL(s(n))$; in § 4, we discuss the relationship between space complexity and reversal complexity. We give a tape reduction theorem for general multitape Turing machines in § 5. Finally, we give some applications of tape reduction results in §§ 6 and 7; in particular, we show that most common reversal complexity classes have complete languages and that hierarchy theorems exist for reversal complexity classes.

2. Efficient use of reversals. Reversal complexity is very different from time and space complexity. Reversal complexity seems much more "powerful" than time and space complexity. For example, only two reversals on a 2-tape Turing machine are enough to duplicate an arbitrarily large tape segment. In any case, our intuitions about reversal complexity are not as well developed as in the case of space and time. Here we present some techniques for efficiently using reversals on Turing machines.

LEMMA 1 (natural number generation). *Given an integer $r > 0$ in unary, the string $S_r = \bar{0}\#\bar{1}\#\bar{2}\# \cdots \#\overline{2^r-1}\#$ can be generated by a 2-tape Turing machine M making $O(r)$ reversals. Here $\bar{m}$ denotes the binary representation of the integer m of length r, prefixed with zeros if necessary.*

Proof. M first generates the pattern $P_r = (0^r\#)^{2^r}$ on one of its tapes, say T_1. This can be done within $O(r)$ reversals. Using P_r, M generates $(01)^{2^{r-1}}$ on its second tape

T_2, making a constant number of reversals.

Now M uses r *phases*, each making a constant number of reversals, to fix the r bits in each segment of zeros separated by # in P_r. Suppose that at the beginning of the $(i+1)$st $(i \geqq 0)$ phase, M has fixed the first i bits for each segment of P_r on tape T_1, and suppose the string $(0^{2^i}1^{2^i})^{2^{r-i-1}}$ is inductively available on tape T_2. Here the first bit of a segment refers to its least significant bit. Note that the $(i+1)$st bit of the jth segment is exactly the jth bit of string $(0^{2^i}1^{2^i})^{2^{r-i-1}}$. Thus, with only one sweep, M can fix the $(i+1)$st bit for each segment of P_r on tape T_1.

Note that in the $(i+1)$st phase M also needs to know which bit in the jth segment is the $(i+1)$st bit. This is easily solved, for example, by placing a special mark on another track below the ith bit of each segment. These marks are easily updated at each phase.

Using a constant number of sweeps, M can generate the string $(0^{2^{i+1}}1^{2^{i+1}})^{2^{r-i-2}}$ from string $(0^{2^i}1^{2^i})^{2^{r-i-1}}$ on tape T_2. It is easy to check that at the end of the rth phase, the string $S_r = \bar{0}\#\bar{1}\#\bar{2}\# \cdots 2^r-1\#$ is on tape T_1 of M. M will terminate when it finds that all the bits have been fixed.

This completes the proof. □

A string of the form $x_1\#x_2\# \cdots \#x_n\#$ $(n \geqq 1,\ x_i \in \Sigma^*,\ \# \notin \Sigma)$ is called a *list* of n *items* (x_i is the ith item). A list is said to be in *normal form* if n is a power of two and each x_i has the same length. The next lemma shows how to convert any list into one in normal form.

LEMMA 2 (list normalization). *There is a 2-tape Turing machine M which, given a list $x_1\#x_2\# \cdots \#x_n\#$ of n items on its input tape, can construct another list $y_1\#y_2\# \cdots \#y_{2^k}\#$ in $O(\log n)$ reversals satisfying*:

(a) $2^{k-1} < n \leqq 2^k$;

(b) *Each y_i $(i = 1, \cdots, 2^k)$ has length* $\max_{i=1,\cdots,n} |x_i|$; *and*

(c) *Each y_i $(i = 1, \cdots, n)$ is obtained by padding x_i with zeros and each y_j $(j = n+1, \cdots, 2^k)$ is a string of zeros.*

Proof. First M computes the unary representation $z_0 \in \{0\}^*$ of $\max_{i=1,\cdots,n} |x_i|$ as follows. With $O(1)$ reversals, M initializes tape 1 to have all the odd-numbered items from the original list and tape 2 to have all the even-numbered items. So tapes 1 and 2 contain the lists $x_1\#x_3\#x_5\# \cdots$ and $x_2\#x_4\#x_6\# \cdots$, respectively. In another pass, M compares x_{2i-1} on tape 1 with x_{2i} on tape 2 $(i = 1, 2, \cdots, \lceil n/2 \rceil)$, marking the longer of the two words. In $O(1)$ passes, M can produce a new list $z_1\#z_2\# \cdots \#z_{\lceil n/2 \rceil}\#$ of these marked words. M repeats this process: it splits the list into two with roughly the same number of items, compares and marks the longer item of each comparison, and produces a new list consisting of only the marked items. After $O(\log n)$ reversals, M is left with a list containing only one item. This item has the longest length among the x_i's. It is now easy to construct z_0.

The next goal is to construct a string of the form

$$(z_0\#)^{2^k} \qquad (k = \lceil \log_2 n \rceil).$$

Suppose we already have $(z_0\#)^{2^i}$. Using $x_1\#x_2\# \cdots \#x_n\#$ and $(z_0\#)^{2^i}M$ can compare n with 2^i: if $n \leqq 2^i$ then we are done, otherwise we will construct $(z_0\#)^{2^{i+1}}$ from $(z_0\#)^{2^i}$ with $O(1)$ reversals.

Finally, from $(z_0\#)^{2^k}$, we can easily construct the desired $y_1\#y_1\# \cdots \#y_{2^k}\#$. □

A more complicated problem is computing the transitive closure of a matrix. There are known efficient parallel algorithms for it. Since reversal is intimately related to parallel time, it is not surprising that we can get a bound on the reversal complexity of transitive closure that is close to the corresponding parallel time bound. It turns

out that the key to computing transitive closure is a fast matrix transposition algorithm. Thus, we have Lemma 3.

LEMMA 3 (matrix transposition). *There is a 2-tape Turing machine M, such that, given an $n \times m$ matrix $A = (a_{ij})$, stored in row major form, M can compute the transpose A^T of A, storing A^T in row major form in one of its tapes, within $O(\log(n+m))$ reversals.*

Proof. We may assume that $n = m = 2^k$ for some integer $k \geqq 1$ and that each entry of A has the same length. By Lemma 2, it is clear that M can use $O(\log n)$ reversals to make A satisfy this property.

For each $i = 0, 1, \cdots, k$, and for $j = 1, 2, \cdots, 2^i$, let $A_j^{(i)}$ denote the $n \times 2^{k-i}$ matrix consisting of the

$$((j-1)2^{k-i}+1)\text{st}, ((j-1)2^{k-i}+2)\text{nd}, \cdots, (j2^{k-i})\text{th}$$

columns of A. For example, $A_j^{(k)}$ is the jth column of A and for each i,

$$A = A_1^{(i)} \mid A_2^{(i)} \mid \cdots \mid A_{2^i}^{(i)}.$$

An *i-row representation* of A is a string consisting of the row-major forms of $A_1^{(i)}, A_2^{(i)}, \cdots, A_{2^i}^{(i)}$, separated by "\$," in that order.

Let $A^{(i)}$ denote the i-row representation of A. The lemma follows if we can show how to obtain $A^{(i+1)}$ from $A^{(i)}$ in $O(1)$ reversals. This is because the input is $A^{(0)}$ (the row major form of A) and the desired output is $A^{(k)}$ (the column major form of A).

In order to do the $A^{(i)} \to A^{(i+1)}$ transformation, we need some auxiliary data. For this, define the block pattern:

$$P^{(i)} = ((w_0^{2^{k-i-1}} w_1^{2^{k-i-1}} \#)^n \$)^{2^i} \qquad (i = 0, \cdots, k-1)$$

where $w_0 = 0^s$ and $w_1 = 1^s$, s being the length of an element of matrix A. Each $w_0^{2^{k-i-1}}$ (respectively, $w_1^{2^{k-i-1}}$) "marks" the left (respectively, right) half of the rows of $A_j^{(i)}$, $j = 1, 2, \cdots, 2^i$. $P^{(i)}$ helps us to "separate" the rows of each $A_j^{(i)}$ into "left half" and "right half."

We may inductively assume that each tape has two tracks with the following contents: $A^{(i)}$ is on track 1 and $P^{(i)}$ is positioned directly underneath $A^{(i)}$ on track 2. The case $i = 0$ is easy to initialize in $O(1)$ reversals. A copy of $P^{(i+1)}$ can be made on track 2 of tape 2 using $P^{(i)}$. Now it is easy to see how to obtain $A^{(i+1)}$ on track 1 of tape 2. □

LEMMA 4 (parallel copying lemma). *There is a 2-tape Turing machine M which, given an input $x = 0^n \# x_1 x_2 \cdots x_m$, where $x_i \in \Sigma^*$, $(i = 1, \cdots, m)$, produces string $y = x_1^{2^n} x_2^{2^n} \cdots x_m^{2^n}$ within $O(n)$ tape reversals, where we assume that M can recognize the boundary between blocks x_i and x_{i+1}, $i = 1, \cdots, m-1$.*

Proof. Using $O(n)$ reversals, M can get a string $S = (x_1 x_2 \cdots x_m)^{2^n}$. Note that $x_1^{2^n} x_2^{2^n} \cdots x_m^{2^n}$ is the transpose of S if we regard S as being a $m \times 2^n$ matrix. □

From the last two important lemmas, we immediately get Lemma 5.

LEMMA 5 (matrix multiplication). *There is a 2-tape Turing machine M such that, given two $n \times n$ Boolean matrices $A = (a_{ij})$ and $B = (b_{ij})$ in row major form, M computes their product $AB = (c_{ij})$ in $O(\log n)$ reversals.*

Proof. By Lemma 3, we can obtain the transpose B^T of B within $O(\log n)$ reversals. Let

$$A = (a_{11} \cdots a_{1n} a_{21} \cdots a_{2n} \cdots a_{n1} \cdots a_{nn})$$

and

$$B^T = (b_{11} \cdots b_{n1} b_{12} \cdots b_{n2} \cdots b_{1n} \cdots b_{nn}).$$

By Lemma 4, we can get

$$A_1 = (a_{11} \cdots a_{1n})^n (a_{21} \cdots a_{2n})^n \cdots (a_{n1} \cdots a_{nn})^n$$

and

$$B_1^T = (b_{11} \cdots b_{n1} b_{12} \cdots b_{n2} \cdots b_{1n} \cdots b_{nn})^n$$

within $O(\log n)$ reversals. Then $O(1)$ more reversals will give AB. So $O(\log n)$ reversals are enough for $n \times n$ Boolean matrix multiplication. □

LEMMA 6 (matrix transitive closure). *There is a 2-tape Turing machine M such that, given an $n \times n$ Boolean matrix $A = (a_{ij})$, stored in row major form, M computes the transitive closure A^* of A in $O(\log^2 n)$ reversals.*

Proof. Since $A^* = (E + A)^n$, where E is the identity matrix, we can reduce this problem to $\log n$ matrix multiplications. □

Sorting seems a very important tool for reversal complexity. Here we give an upper bound for sorting.

LEMMA 7 (sorting). *There is a 2-tape Turing machine M which, given a list $x_1 \# x_2 \# \cdots \# x_n \#$ on its input tape, can construct the sorted list $x_{\pi(1)} \# x_{\pi(2)} \# \cdots x_{\pi(n)} \#$ in $O(\log n)$ reversals. Here each item x_i is assumed to have the form $(k_i ¢ d_i)$ where k_i is a binary number representing the sort key and d_i is the data associated with k_i.*

Proof. By Lemma 2, we can assume that the input list is in normal form. We will be implementing a version of the well-known merge sort. The computation will be accomplished in k phases, where $n = 2^k$. To initialize the first phase, we use the parallel copying lemma to construct on tape 1 a string of the form

$$(x_1\$)^n \# (x_2\$)^n \# \cdots \# (x_n\$)^n \#,$$

where $\$ \notin \Sigma \cup \{\#\}$, using $O(\log n)$ reversals.

For $i = 1, 2, \cdots, k$, let $f(i) = 2^k - \sum_{j=1}^{i-1} 2^j = 2^k - 2^i + 2$. So $f(1) = 2^k$ and $f(k) = 2$. At the beginning of the $(i+1)$st phase $(i = 0, 1, \cdots, k-1)$, we will inductively have, on tape 1, a string of the form

$$B_1 \# B_2 \# \cdots \# B_{2^{k-i}} \#$$

where each B_j has the form

$$(x_{j,1}\$)^{f(i+1)} \% (x_{j,2}\$)^{f(i+1)} \% \cdots \% (x_{j,2^i}\$)^{f(i+1)}$$

and where $x_{j,1}, x_{j,2}, \cdots, x_{j,2^i}$ are distinct items from the original list, already sorted in nondecreasing order. Call B_i the ith *block*. The substring in a block between two consecutive "%'s" is called a *subblock*. Observe that phase 1 has been properly initialized above.

We also need some auxiliary data to carry out the induction. Let $w_0 = 0^s 1$ where $s = |x_i|$ (for all i). For this, we assume that at the beginning of the $(i+1)$st phase, we also store on tape 1 the patterns

$$P_i = (w_0^{2^i} \#)^n \quad \text{and} \quad Q_i = (w_0^{f(i)} \#)^n.$$

Note that Q_i can be easily obtained from P_{i-1} and Q_{i-1} within $O(1)$ reversals and that P_i can be obtained from P_{i-1} in another $O(1)$ reversals. Again, phase 1 is easily initialized.

Let us now see how we carry out phase $i+1$. First we split the string $B_1 \# B_2 \# \cdots \# B_{2^{k-i}} \#$ into two lists containing the odd- and even-numbered blocks:

tape 1 now contains $B_1\#B_3\#\cdots\#B_{2^{k-i}-1}\#$ and tape 2 contains $B_2\#B_4\#\cdots\#B_{2^{k-i}}\#$. This can be done using $O(1)$ reversals.

Our goal is to "merge" B_{2j-1} with B_{2j}, for all $j=1,2,\cdots,2^{k-i-1}$ in parallel. Let

$$B_{2j-1}=(y_1\$)^{f(i+1)}\%(y_2\$)^{f(i+1)}\%\cdots\%(y_{2^i}\$)^{f(i+1)}$$

and

$$B_{2j}=(z_1\$)^{f(i+1)}\%(z_2\$)^{f(i+1)}\%\cdots\%(z_{2^i}\$)^{f(i+1)}.$$

We begin by comparing the first copy of y_1 with the first copy of z_1. Suppose $y_1\geqq z_1$ ("$\geqq$" here is the ordering on the items, as defined by the sort key). Then we "mark" the first copy of z_1 and move head 2 to the first copy of z_2 in the block B_{2j}. We can compare the second copy of y_1 with this copy of z_2, marking the smaller of y_1 and z_2. If y_1 is smaller, we move to the first copy of y_2 and next compare y_2 with z_2. Otherwise, z_2 is smaller and we next compare the third copy of y_1 with the first copy of z_3. In general, suppose we compare (some copy of) y_i with (some copy of) z_j. We mark the smaller of the two. If y_i is smaller, we next move head 1 to the first copy of y_{i+1} and move head 2 to the next copy of z_j and proceed with comparing these copies of y_{i+1} and z_i. Otherwise, z_j is smaller and the roles of y and z are exchanged in the description. (For correctness, we will show below that there is a sufficient number of copies of each item.)

Eventually, one of the blocks is exhausted. At that point, we scan the rest of the other block and mark the first copy of each remaining item. Note that each subblock now contains exactly one marked copy. We then proceed to the next pair of blocks (B_{2j+1} and B_{2j+2}).

After we have compared and marked all the pairs of blocks, we will get two strings S_1 and S_2 (with one copy of an item in each subblock of each block marked) on the two tapes, respectively. Our goal is to produce the merger of B_{2j-1} and B_{2j} from S_1 and S_2. Call the merged result B'_j. We will scan S_1 and output on tape 2 a partially instantiated version of B'_j. Our preceding marking algorithm ensures that if a marked copy of item w in S_1 is preceded by h copies of w, then w has been compared to and found larger than h other items in S_2. In the merge result, we want to place these h smaller items before w. Since each item should be repeated $f(i+2)$ times in its subblock in B'_j we must precede w with

$$h(1+|z_1\$|f(i+2))$$

blank spaces to accommodate these h subblocks, which will be placed there in another pass. With the help of the pattern Q_{i+2}, we can leave the required amount of blank spaces for each subblock in B'_j. More precisely, we make a copy of Q_{i+2} in a track of tape 2 and use it as a "template," which has the block and subblock structure already marked out. As we scan string S_1, for each unmarked copy of an item preceding its marked version, we skip a subblock of Q_{i+2}. When we reach a marked version of item w in S_1, we will copy the $f(i+2)$ successive copies of w into a subblock of Q_{i+2}. To see that there are enough copies of w on S_1 following this marked w, observe that there are a total of $f(i+1)$ copies of w in its subblock in S_1, and since at most 2^i copies of w precede its marked version, there are at least $f(i+1)-2^i\geqq f(i+2)$ copies left. When we finish this process, we scan the partially formed B'_j and the string S_2 simultaneously, and fill in the remaining subblocks of B'_j. We finally have a new list of blocks:

$$B'_1\#B'_2\#\cdots\#B'_{2^{k-i-1}}\#$$

where each B_j' has the required form

$$(x_{j,1}\$)^{f(i+2)}\%(x_{j,2}\$)^{f(i+2)}\% \cdots \%(x_{j,2^{i+1}}\$)^{f(i+2)}.$$

This completes the proof. □

For some applications, we need the sorting algorithm to be stable. Thus, we have the following corollary.

COROLLARY (stable sorting). *The above sorting algorithm can be made stable in this sense: if the output list is*

$$x_{\pi(1)}\#x_{\pi(2)}\# \cdots \#x_{\pi(n)}\#$$

where each $x_{\pi(i)}$ has the form $(k_{\pi(i)}¢d_{\pi(i)})$, then $k_{\pi(i)}=k_{\pi(i+1)}$ implies $\pi(i)<\pi(i+1)$ for all $i=1,\cdots,n-1$.

Proof. To do this, we first number, in another track (say, track K) of the tape, the items in the string to be sorted sequentially. This takes $O(\log n)$ reversals. Next, use the method of Lemma 7 to sort the string. After sorting, those items with a common sort key will form a contiguous block. We apply the sorting method again to each block of items, where we now sort each block by the numbers of items (as written on track K). To do this in $O(\log n)$ reversals, we must do this second sorting for all the blocks simultaneously (our above method can be modified for this). □

3. Simulating deterministic space by reversal. In this section, we will prove one of our main results: for reversal-constructible $s(n)=\Omega(\log n)$,

$$DSPACE(s)\subseteq DREVERSAL_2(s).$$

Thus, for deterministic Turing machines, reversal as a complexity resource is at least as powerful as space. This is a direct consequence of the following theorem.

THEOREM 8. *Suppose $s(n)$ is reversal-constructible and $s(n)=\Omega(\log n)$. Then any language $L\in DSPACE(s)$ can be accepted by a 2-tape deterministic Turing machine M within $O(s)$ reversals.*

Proof. First we assume that $s(n)=\Omega(n)$.

Suppose a deterministic Turing machine N accepts L in space $s(n)$. We construct a 2-tape Turing machine M that accepts L in $O(s(n))$ reversals.

(a) Given an input x of length n, M first generates a string of the form

$$W_1=\bar{0}\#\bar{1}\#\bar{2}\# \cdots \#\overline{2^{s(n)}-1}\#.$$

Since $s(n)$ is reversal-constructible, Lemma 1 implies that this can be done in $O(s(n))$ reversals. Note that we can suppose that all the possible configurations of N on input x are included in the string W_1.

(b) From string W_1, within $O(1)$ reversals, we construct another string

$$W_2=z_0\#z_1\# \cdots \#z_{2^{s(n)}-1}\#$$

such that $|z_m|=s(n)$ and z_m is the "successor configuration" of configuration $\bar{m}$ in the string W_1, $m=0,1,\cdots,2^{s(n)}-1$, as follows. First we scan the string W_1 from left to right. When we have completed scanning a configuration $\bar{m}$ in the string W_1, we know what the "next action" of this configuration of N is, and hence we can put this information at the right end of the corresponding configuration $\bar{m}$ in the string W_1 (for this, we only need to make the alphabet of M large enough). Then we go back through the string W_1 from right to left, using the information recorded in the last scan, to obtain W_2.

Combining strings W_1 and W_2 into one tape, we get a string

$$W_3=u_1\#u_2\# \cdots \#u_{2^s}$$

with each u_m $(m=1,2,\cdots,2^s)$ is a 2-track string of the form

C_m
C'_m

where C_m and C'_m are configurations of N on input x such that $C_m \vdash C'_m$.

(c) Next we construct two identical strings S_1 and S_2 on tapes T_1 and T_2 of M, respectively, using $O(s)$ reversals:

$$S_1=S_2=((u_1\$)^{2^s}\#(u_2\$)^{2^s}\#\cdots\#(u_{2^s}\$)^{2^s}\%)^{2^s}.$$

We will call each substring of the form $(u_1\$)^{2^s}\#(u_2\$)^{2^s}\#\cdots\#(u_{2^s}\$)^{2^s}\%$ a *segment* of S_1 or S_2, and for each $j=1,2,\cdots,2^s$ we will call the substring $(u_j\$)^{2^s}$ a *u_j-subsegment.*

(d) With these two strings, we now find the computation path

$$P=C_1\vdash C_2\vdash\cdots\vdash C_r$$

of N on input x ($r=2^s$ or C_r is terminal) in another $O(1)$ reversals as follows.

(d1) Consider the first u_{i_1}-subsegment in S_1 where the upper track of u_{i_l} contains the initial configuration C_1 of N on input x. Begin by marking the first copy of u_{i_1} in this subsegment and place the head H_1 of T_1 between the first and second copies of u_{i_1} in the u_{i_1}-subsegment. (By "marking" of a copy of u_{i_1}, we mean a special symbol is placed at the end of that copy on a separate track.) Moreover, place the head H_2 of T_2 at the beginning of the string S_2.

(d2) Inductively, suppose we have already found and marked a sequence $u_{i_1}, u_{i_2},\cdots,u_{i_q}$ on the string S_1 on tape T_1 such that the lower track of u_{i_j} is identical to the upper track of $u_{i_{j+1}}$, $j=1,2,\cdots,q-1$. Hence, the upper track of u_{i_l} contains the configuration C_l in the path P, $l=1,2,\cdots,q$. Suppose also that head H_1 is placed between the first and second copies of u_{i_q} in the qth segment in S_1 and that head H_2 is placed at the beginning of the qth segment in S_2. Our goal is to find a subsegment $(u_{i_{q+1}}\$)^{2^s}$ in the qth segment of S_2 such that the lower track of u_{i_q} is identical to the upper track of $u_{i_{q+1}}$. To do this, we compare the lower track of successive copies of u_{i_q} in the subsegment $(u_{i_q}\$)^{2^s}$ of the qth segment of S_1 with the upper track of the first copy of the configuration in each subsegment of the qth segment of S_2.

Eventually, we will find the desired subsegment $(u_{i_{q+1}}\$)^{2^s}$ in the qth segment of the string S_2. The head H_2 will now be between the first and second copies of $u_{i_{q+1}}$ in subsegment $(u_{i_{q+1}}\$)^{2^s}$. We will use the 2^s-1 unscanned copies of this subsegment in S_2 to find the same subsegment $(u_{i_{q+1}}\$)^{2^s}$ in the $(q+1)$st segment in the string S_1. When we find that $u_{i_{q+1}}$-subsegment in S_1, the head H_1 will be between the first and second copies of $u_{i_{q+1}}$ and we move the head H_2 to the beginning of the $(q+1)$st segment in string S_2. Now we are ready for the next iteration.

The fact that there are 2^s copies of u_j in each u_j-subsegment $(j=1,2,\cdots,2^s)$, and there are 2^s segments in each of the strings S_1 and S_2 ensures that we will always have enough copies of each u_j $(j=1,2,\cdots,2^s)$ for the comparisons in our algorithm.

If N accepts or rejects x, then at some moment in the above iteration, M will realize it. M will accept or reject accordingly. If N is in an "infinite loop" then M will exhaust one of the two strings S_1 and S_2 before it finds a terminal configuration of N on input x: M will also reject x in this case.

Finally, we indicate how to modify the above proof for the case where $s(n)=\Omega(\log n)$. Instead of storing the entire input string x with each configuration, we attach

to each modified configuration of N only the position of the input head. Then by sorting the string W_1 by the positions of the input head, followed by one sweep of the input x, we are able to record the "current input symbol" into each of the modified configurations. A similar sort can be applied to W_2 in its construction.

This completes the proof. $\square$

We should remark that a result of Rytter and Chrobak [10] may appear to be stronger than Theorem 8 for the case $s(n) = \log n$. Their result says that $O(1)$ reversals are sufficient to simulate $O(\log n)$ space. However, a closer examination reveals that they do not count reversals on the input tape—in fact their simulation makes a polynomial number of reversals on the input tape.

4. The relationship between space and reversal. Borodin [3] proved that an $s(n)$ space-bounded computation of nondeterministic multitape Turing machines can be simulated by uniform circuits of depth $O(s^2(n))$. Later, Pippenger [9] showed that uniform circuits of depth $d(n)$ can be simulated by $O(d^3(n))$ reversal-bounded deterministic multitape Turing machines. However, Borodin and Pippenger were using two different kinds of uniform circuits: Borodin's circuits are "space uniform," while Pippenger's circuits are "reversal uniform." A closer investigation reveals that Borodin's circuits are in fact "reversal uniform" as well. Thus, combining their results with a slight modification, we obtain

$$NSPACE(s) \subseteq DREVERSAL(s^6).$$

This yields the strongest result for the relationship between space and reversal complexity until now.

Using Theorem 8, we can greatly improve this result in Theorem 9.

THEOREM 9. *Suppose $s(n)$ is reversal-constructible and $s(n) = \Omega(\log n)$. Then any language $L \in NSPACE(s)$ can be accepted by a 2-tape deterministic Turing machine M within $O(s^2)$ reversals.*

Proof. By Theorem 8 and Savitch's result [11]:

$$NSPACE(s(n)) \subseteq DSPACE(s^2(n))$$

we get our desired theorem. Note that the reversal constructibility of the function $s(n)$ implies the reversal constructibility of the function $s^2(n)$. $\square$

Remark. In fact, we can give a direct proof for this theorem, using a method different than that used in Theorem 8. For this, we use Lemma 6 to compute the transitive closure of the transition matrix of N on input x.

We can get an even stronger result. Borodin, Cook, and Pippenger [2] have shown the following result:

$$RSPACE(s) \subseteq DSPACE(s^2)$$

where $RSPACE(s)$ denotes the class of languages which are accepted by probabilistic Turing machines within $s(n)$ space. Using Theorem 8, we get Theorem 10.

THEOREM 10. *Suppose $s(n)$ is reversal-constructible and $s(n) = \Omega(\log n)$. Then any language $L \in RSPACE(s)$ can be accepted by a 2-tape deterministic Turing machine M within $O(s^2)$ reversals.*

The restriction of constructibility of the function $s(n)$ is not severe. We observe that most common complexity functions are reversal-constructible.

LEMMA 11. *For all integers $k \geqq 1$, n^k ($\log^k n$, k^n, n^n, respectively) is $\log n$ ($\log n$, n, n, respectively)-reversal-constructible.*

The proof of this lemma is omitted.

If we do not want to restrict the function $s(n)$ to be constructible, then we have the following theorem.

THEOREM 12. *Let $s(n)$ be an arbitrary complexity function with $s(n)=\Omega(\log n)$. Then*

$$DSPACE(s) \subseteq DREVERSAL(s^2)$$

and

$$NSPACE(s) \subseteq DREVERSAL(s^3).$$

Proof. The proofs for these two results are similar, so we only outline the proof for the first one.

We use the well-known technique of guessing the value of $s(n)$, starting from $s(n)=\log n, 1+\log n, \cdots$. For each guessed value h, we use $O(h)$ reversals doing the computation in the proof of Theorem 8. If the input is in the language, we will accept when $h=s(n)$, using at most

$$\sum_{h=\log n}^{s(n)} O(h)=O(s(n)^2)$$

reversals. □

To obtain results in which we simulate reversal-bounded Turing machines by space-bounded Turing machines efficiently, we first note the following lemma. (See Yap [13] for a proof.)

LEMMA 13. *If a language L is accepted by a deterministic Turing machine M within reversal $r(n)=\Omega(\log n)$, then M simultaneously accepts within time $O(2^r)$ and space $O(2^r)$.*

THEOREM 14. *For arbitrary $r=\Omega(\log n)$,*

$$DREVERSAL(r) \subseteq DSPACE(r^2).$$

Proof. Simon [12] (see Yap [13]) has shown that $D\text{-}TIME\text{-}REVERSAL(t, r) \subseteq DSPACE(r \log t)$. By Lemma 13, we get the conclusion immediately. □

We close this section with the following important corollary.

COROLLARY.

$$DSPACE(n^{O(1)}) = NSPACE(n^{O(1)}) = DREVERSAL(n^{O(1)}),$$

$$DSPACE(\log^{O(1)} n) = NSPACE(\log^{O(1)} n) = DREVERSAL(\log^{O(1)} n).$$

5. Tape reduction theorem for reversal complexity. Now we are ready for the tape reduction result for reversal complexity. In fact, if we restrict the complexity functions to be reversal-constructible, then from Theorems 14 and 8, we get immediately the following.

If $r(n)=\Omega(\log n)$ and $r^2(n)$ is reversal-constructible, then

$$DREVERSAL(r) \subseteq DREVERSAL_2(r^2).$$

However, here we would like to remove the restriction of constructibility of the complexity functions and give a direct proof for general complexity functions. For this, we introduce the concept of "*phase.*"

To define "phase" we first give a precise definition of reversal complexity. Let $\bar{C}=(C_i)_{i\geqq 0}$ be a computation path of a k-tape Turing machine. We say that head h ($h=0, \cdots, k$) *tends in direction* d (for $d \in \{-1, +1\}$) in C_i if the last transition $C_{j-1} \vdash C_j$ ($j \leqq i$) preceding C_i in which head h moves is in the direction d. If head h has been

stationary up until C_i we say head h tends in the direction $d=0$ in C_i. We also say that head h has *tendency* d if it tends in direction d. It is important to note that the head tendencies of a configuration C are relative to the computation path in which C is embedded. We say head h makes a *reversal* in the transition $C_{j-1} \vdash C_j$ in $\bar{C}$ if the tendency of head h in C_{j-1} is *opposite* to its tendency in C_j. We say $C_{j-1} \vdash C_j$ is a *reversal* (transition). The *reversal* of $\bar{C}$ is the total number of reversals made by all the heads over the entire computation path.

Observe that changing the tendency from 0 to ± 1 is not normally regarded as a reversal. However, for the purposes of some proofs, we may regard $C_{j-1} \vdash C_j$ to be a reversal if the tendencies of some head h in C_{j-1} and in C_j are *different*: this only overcounts the number of reversals by $k+1$—once for each of the heads $h=0, \cdots, k$. Each computation path $\bar{C}$ can be uniquely divided into a sequence of disjoint partial computation paths $P_0, P_1, \cdots$, where every configuration in P_i ($i=0,1,\cdots$) has the same head tendencies. Each P_i is called a *phase*. If $C_i \vdash C_{i+1} \vdash \cdots \vdash C_{i+r}$ is a phase, then C_i and C_{i+r} will be called the *start configuration* and *end configuration* of the phase, respectively. The transition from one phase P_i to the next P_{i+1} is caused by the reversal of at least one head. Clearly, the reversal of $\bar{C}$ is bounded by $k+1$ times the number of phases in $\bar{C}$.

LEMMA 15. *If a language L is accepted by a Turing machine N making at most $r(n)$ reversals on input of length n, then there is a Turing machine M which satisfies the following conditions*:

(1) *M and N have the same number of tapes*;

(2) *M accepts L in at most $r(n)$ reversals*;

(3) *in each step of M, at least one tape head moves*; *and*

(4) *The head of each work tape of M does not change the symbol it is scanning unless it is about to leave the square.*

Proof. Given a Turing machine N accepting L, by increasing the "finite state complexity" of N, we can obtain the Turing machine M satisfying the conditions stated in the lemma. □

THEOREM 16 (tape reduction for reversals). *Let $r(n)$ be any complexity function where $r(n)=\Omega(\log n)$. If L is accepted by a multitape Turing machine within $r(n)$ reversals, then it is accepted by a 2-tape Turing machine M that makes at most $O(r^2)$ reversals, i.e.,*

$$DREVERSAL(r) \subseteq DREVERSAL_2(r^2).$$

Proof. We will use a direct simulation.

Suppose L is accepted by a k-tape Turing machine N within $r(n)$ reversals on input of length n. Without loss of generality, we assume that N satisfies the properties stated in Lemma 15.

We will construct a 2-tape Turing machine M which will simulate N and make at most $O(r^2(n))$ reversals.

M uses $k+1$ tracks on one of its tapes, say, T_1, to hold the configurations of N, namely, track 0 corresponds to the input tape of N and track j ($1 \leqq j \leqq k$) corresponds to the tape j of N. Sometimes we also talk about "the head position" or "the head tendency" of track j, referring to the corresponding simulated head of N.

For the start configuration of phase i in the computation of N, M can, within $O(1)$ reversals, make the corresponding configuration of T_1 satisfy the following properties:

(1) All heads of the tracks of T_1 will tend to the right in that phase;

(2) All heads of the tracks of T_1 are at the square 0.

So the start configuration of each phase of N is represented on the tracks of T_1 as follows:

track 0	$x\ x\ x\ x\ x$	$\uparrow\to x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x$
track 1	$x\ x\ x$	$\uparrow\to x\ x\ x\ x\ x\ x$
$\cdots$	$\cdots\cdots$	$\cdots\cdots\cdots\cdots\cdots\cdots$
track k	$x\ x\ x\ x$	$\uparrow\to x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x$

where $\uparrow$ stands for the position of the head of the track and $\to$ indicates the tendency of the tape head. Note that the contents of tape i of N may be written in reverse order on track i of T_1.

By a *full trace* of phase i of N, we mean a tuple of the form

$$\tau=\langle q, h_0, a_0, \cdots, h_k, a_k\rangle$$

where q is a state of N, a_j's are tape symbols of N, and the h_j's lie in the range $0\leqq h_j<c^i$, where c is a constant depending on N. Intuitively, h_j denotes a head position on tape j of N and a_j is the symbol under the head. Lemma 13 shows that, within the ith phase, no head position h_j can be out of the range $0\leqq h_j<c^i$. We will store each full trace in $2k+3$ tracks of tape T_2 of M, with the first $k+1$ tracks for the h_j's, the $k+2$nd track for the q, and the last $k+1$ tracks for the a_j's.

A *partial trace* of phase i of N is obtained by replacing any of the components of a full trace by a distinguished symbol $*$.

Let $\mathcal{K}_0^i$ be the set of all partial traces of the ith phase of N. We define a *quasi ordering* on $\mathcal{K}_0^i$ as follows. If

$$\tau=\langle -, h_0, -, \cdots, h_k, -\rangle$$

where $-$ means that we do not care what is in that position, and if

$$\tau'=\langle -, h_0', -, \cdots, h_k', -\rangle,$$

then $\tau\leq\tau'$ if and only if $h_l\leqq h_l'$ for $l=0, \cdots, k$. If $\tau\leq\tau'$ and $\tau'\leq\tau$, then we say that they are equivalent, and write $\tau\asymp\tau'$. (This is a quasi ordering rather than a partial ordering because $\tau\asymp\tau'$ does not mean $\tau=\tau'$.)

M will simulate phase i of N using the following steps (details for accomplishing these steps follow afterwards).

Step 1. M generates on the tape T_2 all partial traces of the ith phase of N which has the form

$$\langle q, h_0, *, \cdots, h_k, *\rangle,$$

i.e., all the tape symbols are replaced by "$*$." Let $W_1^{(i)}$ denote this list of partial traces.

Step 2. Next, M converts each partial trace in $W_1^{(i)}$ into a full trace by filling in the correct tape symbol for each head position on each track of the trace. More precisely, if h_l is the head position of N on track l, and a_l is the symbol in the h_lth square on track l of tape T_1, then we place a_l into the $(k+2)+l$th track in the tape T_2 under the h_l. Let $W_2^{(i)}$ denote this list of full traces.

Step 3. M does a "topological sort" of the list $W_2^{(i)}$ of full traces found on tape T_2. By this we mean that if τ and τ' are full traces ($\tau\leq\tau'$ and τ and τ' are not equivalent), then τ appears before τ'.

Step 4. For each full trace τ in the list $W_2^{(i)}$, construct its successor partial trace τ'. We place τ' on another $2k+3$ tracks directly below τ on the tape T_2. By the successor partial trace τ' of full trace τ, we mean that if τ corresponds to a

configuration C of N and $C \vdash C'$ in phase i, then τ' corresponds to the configuration C'.

Step 5. Similar to the proof of Theorem 8, using the string on tape T_2 obtained in Step 4, we can place marks on the upper $2k+3$ tracks of the tape T_2 to determine a sequence P_i of full traces

$$\tau_0 \vdash \tau_1 \vdash \cdots \vdash \tau_r$$

corresponding to the partial computation path in phase i of the computation of N, where τ_0 and τ_r, respectively, are the full traces corresponding to the start and end configurations of phase i.

Step 6. By the definition of the quasi ordering, we have

$$\tau_0 \preceq \tau_1 \preceq \cdots \preceq \tau_r$$

and they appear in this order in the tape T_2, so we can read the P_i sequentially on the tape T_2 and modify the contents of each track of tape T_1 accordingly.

Step 7. Prepare T_1 for the next phase of N.

M will repeat the above algorithm until it finds that N accepts or rejects; then M accepts or rejects accordingly.

Clearly, M accepts if and only if N accepts.

Now we present the details to make sure that M makes at most $O(r^2)$ reversals.

Step 1. By Lemma 13 we can assume that h_j is a c-ary number, so its length $|h_j|$ is less than or equal to i, for all $j = 0, 1, \cdots, k$. Similar to Lemma 1, we can generate the string

$$S_i = 0_c \# 1_c \# \cdots \# (c^i - 1)_c$$

within $O(i)$ reversals, where m_c is the c-ary representation of integer m of length i, prefixed with zeros if necessary.

From S_i, we can construct $k+1$ strings defined as follows:

$$S_i^{(0)} = S_i^{c^{ik}}$$

and

$$S_i^{(h-1)} = ((0_c \#)^{c^{ih}} (1_c \#)^{c^{ih}} \cdots ((c^i - 1)_c \#)^{c^{ih}})^{c^{i(k-h)}},$$

$h = 1, \cdots, k$. By Lemma 4, each of these $S_i^{(h-1)}$'s can be constructed within $O(\log(c^{ik})) = O(i)$ reversals.

If we put string $S_i^{(h+1)}$ in the track $h+1$ of T_2, $h = 0, 1, \cdots, k$, then in T_2 we will get a string

$$I_1 = K_1 \# K_2 \# \cdots \# K_{c^{i(k+1)}}$$

where all K_j ($j = 1, \cdots, c^{i(k+1)}$) are different, and each K_j is of form

$$\langle (h_0)_c, (h_1)_c, \cdots, (h_k)_c \rangle,$$

$0 \leqq h_l < c^i$, (with $(h_l)_c$ on track l), $l = 0, 1, \cdots, k$.

Suppose N has e states. From I_1 we construct

$$I_2 = K_1^e \# K_2^e \# \cdots \# K_{c^{i(k+1)}}^e.$$

We construct $W_1^{(i)}$ on tape T_2 by putting a different state of N to the $(k+1)$st track of each K_j in K_j^e in I_2, $j = 1, \cdots, c^{i(k+1)}$.

$W_1^{(i)}$ can now be represented by the following string:

$$W_1^{(i)} = \tau_1 \# \tau_2 \# \cdots \# \tau_{ec^{i(k+1)}}$$

where each τ_j is of form

$$\langle q, (h_0)_c, *, \cdots, (h_k)_c, *\rangle,$$

with $(h_l)_c$ in the lth track, $l = 0, 1, \cdots, k$, and q in the $(k+1)$st track of τ_j.

Step 2. It is enough to show how we deal with one track, say track j, of T_2. First, we sort the partial traces in the list $W_1^{(i)}$ on tape T_2 by the head positions on track j. This can be done within $O(i)$ reversals. Then using one more reversal, we can scan tapes T_1 and T_2, reading the corresponding symbol on the h_jth square in the track j in tape T_1, putting the symbol into the track $(k+2)+j$ under the corresponding head position $(h_j)_c$ in tape T_2, $h_j = 0, 1, \cdots, c^i - 1$.

Step 3. To topologically sort the full traces in the list $W_2^{(i)}$ on tape T_2, we only have to do a stable sort of the full traces by the head position on track 0, then by the head position on track $1, \cdots$, finally by the head position on track k. This can be done in $O(i)$ reversals. We claim that if $\tau \leq \tau'$ and τ and τ' are not equivalent, then τ appears before τ' in the list $W_2^{(i)}$. In fact, if τ' appears before τ in the list $W_2^{(i)}$, then at least on one of the tracks, the head position of τ is "larger" than the head position of τ'. Thus, $\tau \leq \tau'$ cannot be true.

Step 4. For each of the full traces in the list $W_2^{(i)}$, we can decide immediately what the "next" action is. That is, given a full trace

$$\tau = \langle q, h_0, a_0, \cdots, h_k, a_k\rangle,$$

we can decide the successor partial trace

$$\tau' = \langle q', h_0', *, \cdots, h_k', *\rangle.$$

Thus, by scanning $W_2^{(i)}$ within $O(1)$ reversals, we can construct a list $W_3^{(i)}$ such that the lth partial trace of $W_3^{(i)}$ is the successor partial trace of the lth full trace of $W_2^{(i)}$ in phase i for all l.

(Remark: Suppose $\tau \vdash \tau'$ in phase i, where both τ and τ' are full traces; then the head positions of τ' will be obtained by adding the corresponding head positions of τ by one or zero. If a full trace does not have successor partial trace in phase i (for example, it corresponds to the end configuration of phase i), we just designate a special symbol to denote its symbolic successor.)

So Step 4 can be done in $O(i)$ reversals.

Step 5. Note that unlike the proof of Theorem 8, the current symbol of each trace is uniquely determined by the head position, so there is no need to convert the partial traces of $W_3^{(i)}$ to full traces. Hence, we can compare the full traces in the list $W_2^{(i)}$ with the partial traces in the list $W_3^{(i)}$ directly. As in theorem 8, we use parallel copying, mark certain traces, etc.

Step 6. Now collect into a contiguous list $W_4^{(i)}$ all marked full traces in the list $W_2^{(i)}$, in the order in which they appear in list $W_2^{(i)}$. Note that this is exactly the order in which the corresponding configurations of N appear in the partial computation path in phase i. Now it should be clear how we modify the tape contents of T_1 to obtain the contents which represent the start configuration of the next phase of N: For each track h, by scanning the list $W_4^{(i)}$, we would know how to update the symbols of that track from left to right. Note that $O(1)$ reversals suffice for all the tracks.

Step 7. We reverse the contents on those tracks of tape T_1 that corresponds to tape heads of N, making a reversal as we go from phase i to phase $i+1$.

Since N has less than or equal to $r(n)$ phases and M needs at most $O(i)$ reversals to simulate the ith phase of N, M makes $O(r^2(n))$ reversals. This completes our proof. □

6. Reversal hierarchies. Now that we have a tape reduction theorem for reversal complexity, we can obtain reversal analogues of well-known theorems on space and time complexity. In this section, we prove a hierarchy result for reversal classes.

THEOREM 17 (reversal complexity hierarchy). *Let $r_1(n)$ and $r_2(n)$ be complexity functions such that $(r_2(n))^2 = o(r_1(n))$, $r_2(n) = \Omega(\log n)$, and $r_1(n)$ is reversal-constructible; then*

$$DREVERSAL(r_1(n)) - DREVERSAL(r_2(n)) \neq \emptyset.$$

Proof. Since $(r_2(n))^2 = o(r_1(n))$, by the deterministic space hierarchy theorem, there exists a language L such that

$$L \in DSPACE(r_1(n))$$

and

$$L \notin DSPACE((r_2(n))^2).$$

By Theorem 8,

$$L \in DREVERSAL(r_1(n)),$$

and by Theorem 14,

$$L \notin DREVERSAL(r_2(n)).$$

This gives

$$L \in DREVERSAL(r_1(n)) - DREVERSAL(r_2(n)). \qquad \square$$

Remark. We can prove this theorem directly with the help of the tape reduction theorem, but an appeal to the space hierarchy theorem as above gives a shorter proof.

7. Complete languages for reversal complexity. First we prove Lemma 18.

LEMMA 18. *Let M be a Turing machine that accepts in $f(n)$ reversals. If $f(n)$ is reversal-constructible, then there exists a Turing machine N such that*

(1) *N and M have the same number of tapes*;

(2) *$L(M) = L(N)$*;

(3) *N makes at most $O(f(n))$ reversals*; *and*

(4) *N always halts.*

Proof. Since $f(n)$ is reversal-constructible, N first computes the unary form of $f(n)$, appending it to the end of one of its tapes. N will use it as a counter to protect itself from making too many reversals. Then N simulates M step by step, and in its finite control N guards the "infinite loop" of M. In this description, by "infinite loop" we mean that M is in a nonhalting computation path in which no reversal is made. Once N finds out that M is using too many reversals or is in an "infinite loop," N rejects immediately. It is easy to see that N satisfies the properties stated in the theorem. $\square$

COROLLARY. *If $f(n)$ is reversal-constructible, then for any $h \geqq 2$*

$$co\text{-}DREVERSAL_h(f(n)) = DREVERSAL_h(f(n)).$$

As noted in Yap [13], hierarchy theorems and the existence of complete languages for a complexity class can be proved in a systematic fashion using the concept of "efficient universal machines," defined in the following way.

DEFINITION. A Turing machine U is an *efficient universal machine* for a class $DREVERSAL(F)$, where F is a family of complexity functions, if

(1) U takes input of the form $i\#x$ on its input tape where $i, x \in \{0, 1\}^*$;

(2) Let $L_i(U)=\{x \mid U \textit{ accepts } i\#x\}$; then

(2.1) For all $i \geqq 0$, there exists an $f \in F$ such that U accepts $i\#x$ within $O_i(f(|x|))$ reversals, where the subscript i means that the implicit constant in the O-notation depends on i;

(2.2) For any $L \in DREVERSAL(F)$, where $L \subseteq \{0, 1\}^*$, there are infinitely many indices i such that $L = L_i(U)$.

For a general universal machine U (not necessarily efficient) taking input of the form $i\#x$, if we fix the prefix $i\#$ of the input, we can conveniently regard U as an ordinary Turing machine U_i which accepts the language $L_i(U)=\{x \mid U \textit{ accepts } i\#x\}$. Sometimes we write $\{U_0, U_1, \cdots\}$ instead of U.

We now have the following theorem.

THEOREM 19. *The classes* $DREVERSAL(n^{O(1)})$ *and* $DREVERSAL(O(1)^n)$ *have efficient universal machines.*

Proof. We just show the result for $DREVERSAL(n^{O(1)})$.

Let $U=\{U_0, U_1, \cdots\}$ be the universal Turing machine for 2-tape deterministic Turing machines. We construct a new 3-tape Turing machine $\bar{U}$ as follows.

For a given input $i\#x$, $\bar{U}$ simulates U_i on input x step by step for at most $|x|^i$ reversals. If U_i does not accept x within $|x|^i$ reversals, $\bar{U}$ rejects, otherwise $\bar{U}$ accepts. To simulate U_i, $\bar{U}$ first makes $O(2^{|x|^i})$ copies of the "machine code" i for U_i and puts them on its tape, say, T_1. This can be done within $O_i(|x|^i)$ reversals by Lemmas 4 and 11.

Then, $\bar{U}$ can simulate each step of U_i by first reading a copy of the "machine code" i of U_i on tape T_1 without making extra reversals. $O(2^{|x|^i})$ copies of machine code are enough for the whole simulation, by Lemma 13.

It is easy to check that $\bar{U}$ is an efficient universal machine for $DREVERSAL(n^{O(1)})$. By Lemma 18, we can suppose that $\bar{U}$ on any input $i\#x$ always halts and makes $O_i(|x|^i)$ reversals. If a language L is in $DREVERSAL(n^k)$ then our Theorem 16 shows that $L \in DREVERSAL_2(n^{2k})$, so $\bar{U}$ can recognize L efficiently. □

THEOREM 20. *The classes* $DREVERSAL(n^{O(1)})$ *and* $DREVERSAL(O(1)^n)$ *have complete language under* log-*space many-one reducibility.*

Proof. Again, let us consider the case of $DREVERSAL(n^{O(1)})$.

From Theorem 19, we may suppose that $U=\{U_1, U_2, \cdots\}$ is the efficient universal Turing machine for $DREVERSAL(n^{O(1)})$, where all U_i always halt on any input, U_i accepts or rejects in $f_i(n)$ reversals and $f_i(n)$ is a polynomial.

It is easy to check that the following language is $DREVERSAL(n^{O(1)})$-complete:

$$L=\{w=i\#x\#0^m\colon m=|x|^i,\ x \in U_i\}. \qquad \square$$

8. Remarks. In this paper, we proved various relationships between space complexity and reversal complexity, and also gave a fundamental tape reduction theorem for reversal complexity. We demonstrated that deterministic reversal and space are polynomially related. If we accept that reversal is parallel time, then we have yet another evidence of the so-called "Parallel computation thesis." We suggest some directions for further research.

(1) Can our result that

$$DREVERSAL(r) \subseteq DREVERSAL_2(r^2)$$

be improved to either

$$(*) \qquad DREVERSAL(r) \subseteq DREVERSAL_1(r^k)$$

or

$$(**) \qquad DREVERSAL(r) \subseteq DREVERSAL_h(r)$$

for some constants k and h? We believe $(*)$ is unlikely to be true and $(**)$ seems to require new techniques.

(2) The problem of "speedup reversal complexity by a constant factor" still remains open. It is known that there are languages accepted with k reversals but not with $k-1$ reversals. So any linear speedup result for reversal must exclude the case where a constant number of reversals are made.

REFERENCES

[1] B. S. Baker and R. V. Book, *Reversal-bounded multipushdown machines*, J. Comput. Systems Sci., 8 (1974), pp. 315–322.

[2] A. Borodin, S. Cook, and N. Pippenger, *Parallel computation for well-endowed rings and space-bounded probabilistic machines*, Inform. and Control, 58 (1983), pp. 113–136.

[3] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–743.

[4] P. C. Fisher, *The reduction of tape reversal for off-line one-tape Turing machines*, J. Comput. Systems Sci., 2 (1968), pp. 136–147.

[5] J. Hartmanis, *Tape-reversal bounded Turing machine computations*, J. Comput. Systems Sci., 2 (1968), pp. 117–135.

[6] F. C. Hennie and R. E. Stearns, *Two-tape simulation of multitape Turing machines*, J. Assoc. Comput. Mach., 13 (1966), pp. 533–546.

[7] J.-W. Hong, *On similarity and duality of computation*, in Proc. 21st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1980, pp. 348–359.

[8] T. Kameda and R. Vollmar, *Note on tape reversal complexity of languages*, Inform. and Control, 17 (1970), pp. 203–215.

[9] N. Pippenger, *On simultaneous resource bounds*, in Proc. 20th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1979, pp. 307–311.

[10] W. Rytter and M. Chrobak, *A characterization of reversal-bounded multipushdown machine languages*, Theoret. Comput. Sci., 36 (1985), pp. 341–344.

[11] W. J. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. Systems Sci., 4 (1970), pp. 177–192.

[12] I. Simon, *On some subrecursive reducibilities*, Tech. Report STAN-CS-77-608, Computer Science Department, Stanford University, Stanford, CA, April 1977.

[13] C. K. Yap, *Introduction to the Theory of Complexity Classes*, Oxford University Press, New York, to appear.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 639-654, August 1991

© 1991 Society for Industrial and Applied Mathematics

003

COMPUTING THE STRENGTH OF A GRAPH*

DAN GUSFIELD†

Abstract. The strength of a graph is a measure of its vulnerability, strictly generalizing the edge connectivity of a weighted graph. Cunningham [*J. Assoc. Comput. Mach.*, 32 (1985), pp. 549-562] showed how to compute the strength of a graph in $O(|V|^4|E|)$ time, using ideas from polymatroids and network flow. In this paper, his polymatroid approach is modified, a modified version of the Goldberg-Tarjan network flow algorithm [*J. Assoc. Comput. Mach.*, 4 (1988), pp. 136-146] is used. Then, using ideas developed by Gallo, Grigoriadis, and Tarjan [*SIAM J. Comput.*, 18 (1989), pp. 30-55], and by Gusfield, Martel, and Fernandez-Baca in [*SIAM J. Comput.*, 16 (1987), pp. 237-251], it is shown that the solution runs in $O(|V|^3|E|)$ time. Analogous speedups for sparse-case bounds are also obtainable.

Key words. graph vulnerability, parametric network flow, bipartite flow, polymatroids

AMS(MOS) subject classifications. 90B10, 68R10, 90C27, 68Q25, 05C99

1. Introduction. In a major paper on parametric network flow by Gallo, Grigoriadis, and Tarjan [4], the Goldberg-Tarjan network flow algorithm is shown to have the surprising property that it can solve a sequence of $O(|V|)$ network flow problems of a common important type within the same time bound established for just a single network flow computation; we will refer to this as the GGT method or GGT result. The GGT method is exploited in [4] to speed up (by a factor of between $\log|V|$ and $|V|$) the computation of several well-studied combinatorial problems, firmly establishing the importance of the method in combinatorial computing. Each of the problems considered in [4] involves computing a sequence of flows, and for most of these problems, the Goldberg-Tarjan flow algorithm can be directly applied to the sequence, yielding a speedup over the time needed to compute each flow in the sequence independently. One such problem discussed in [4] is that of computing the *strength of a directed graph*, first considered by Cunningham [3]. However, the major focus of Cunningham's paper is the problem of computing the strength of an *undirected* graph. This problem seems considerably harder[1] than the directed version, as evidenced by the depth of the mathematical ideas used in the original solution of the undirected case compared to the directed case [3], by the intricacy and time complexity of that solution, and by the difficulty in adapting it to the flow methods of GGT.

In this paper we show that the parametric method of GGT can be extended to compute the strength of an undirected graph, achieving a factor of $|V|$ speed up over the solution reported in [3]. However, Cunningham's initial solution does not fit the model specified in [4], and so the GGT methodology cannot be directly applied: both Cunningham's original solution, and the original Goldberg-Tarjan algorithm and analysis must be modified to obtain the speed up, and the method must be applied not just to a single sequence of flows, but to several interleaved sub-sequences of flows. Hence, in addition to accelerating the computation of graph strength, this paper further illustrates the importance of the general GGT methodology, as it shows that there are

* Received by the editors August 24, 1987; accepted for publication (in revised form) September 7, 1990. This research was partially supported by United States Census Bureau grant JSA 86-9 and National Science Foundation grant CCR 8803704.

† Computer Science Division, University of California, Davis, California 95616.

[1] The comparison between the problems in [4] shows that the undirected strength problem is harder than the directed case.

applications of the methodology outside of the original model considered in [4]. More generally, it suggests that additional applications for the GGT methodology can be found by the specific approach taken in this paper: convert a nonbipartite flow problem where internal edges have parameterized capacities to a larger bipartite flow problem where the only parameterized edges are incident with the source or sink node, as is required by the GGT model; then exploit properties of the bipartite network to show that, despite the increased size, flow computation on the bipartite network is still fast.

1.1. Background on the strength of a network.

DEFINITION. $G=(V, E)$ is a connected undirected graph with nonnegative edge weight $s(e)$ on each edge e. The weight $s(e)$ will be called the *strength* of the edge e.

DEFINITION. For a set of edges $A \subseteq E$, let $k(A)$ be the number of additional connected components created by deleting edges A from G. That is, $k(A)$ is one less than the total number of components in $G-A$.

DEFINITION. Let $S(A)$ be the total strength of the edges in A, i.e., $S(A)=\sum_{e \in A} s(e)$.

DEFINITION. The **strength** of graph G, denoted $\sigma(G)$, is defined as $\min(S(A)/k(A))$ taken over all subsets of edges A such that $k(A)>0$.

Strength is the minimum possible cost per new component created, and is a measure of graph vulnerability strictly generalizing the edge connectivity of a weighted graph. Connectivity (which corresponds to the least weight set of edges that must be deleted in order to create *one* new component) is often used as a simple measure of graph vulnerability, or the disruptability of the system that the graph models. Connectivity is clearly not an adequate measure for all purposes, but it is easy to compute. Strength yields more information about vulnerability than does connectivity, and while it is also not a perfect measure, it is useful because it also can be computed efficiently. Strength, as a measure of vulnerability, was first suggested in [8] in the case that all edge weights are one, and in the more general form in [3].

Cunningham [3] first showed that the strength of G, $\sigma(G)$, can be computed in polynomial time with the approach sketched below.

DEFINITION. For a fixed value of parameter b, the **attack** problem for b is that of finding a set A to minimize $[S(A)-bk(A)\colon A \subseteq E]$.

LEMMA (Cunningham [3]). *$\sigma(G)$ equals the maximum value of b such that the set $A=\varnothing$ is an optimal solution of the attack problem for b. Further, $\sigma(G)$ can be computed by solving at most $|V|$ attack problems, where the value of b decreases for each successive problem.*

Hence the key to computing strength is in how to solve a single attack problem. Cunningham reduces the attack problem to $|E|$ network flow problems in a sequence of reductions described below.

DEFINITION. For a set of edges A, define $r(A)$ as the size of the largest subset of A which contains no cycles.

DEFINITION. Let $\bar{A}$ be $E-A$, the complement set of edges of A.

It is easy to prove that $r(A)=|V|-[k(\bar{A})+1]$, hence the attack problem can be solved as $\min[S(A)+br(\bar{A})\colon A \subseteq E]$.

This problem is again transformed by setting $x(e)=s(e)/b$ for each edge e, and then solving the problem $\min[x(A)+r(\bar{A})\colon A \subseteq E]$, where $x(A)=\sum_{e \in A} x(e)$.

DEFINITION. For $B \subseteq V$, let $\gamma(B)$ be the set of edges in the subgraph of G induced by B.

DEFINITION. Let y be an assignment of real values to the $|E|$ edges to G, and for a subset $A \subseteq E$, let $y(A)$ be defined as $\sum_{e \in A} y(e)$.

DEFINITION. Let $P(G)=\{y\geqq 0: y(\gamma(B))\leqq |B|-1$ for all $B\neq\emptyset$, $B\subseteq V\}$.

DEFINITION. Given an $|E|$ length vector x, vector y is said to be *maximal* with respect to x and $P(G)$, if $y\leqq x$ componentwise, $y\in P(G)$, and no component of y can be increased without violating one of those two conditions.

The following theorem and algorithm are due to Edmonds, and are described and justified in [3].

THEOREM (Edmonds). *Given a nonnegative $|E|$ length vector x, let y be any assignment of values to edges of G which is maximal with respect to x and $P(G)$. Then $y(E)=\min[x(A)+r(\bar{A}): A\subseteq E]$. Further, the method described for finding such a maximal y also yields a subset of edges A which minimizes $[x(A)+r(\bar{A}): A\subseteq E]$.*

POLYMATROID GREEDY ALGORITHM (Edmonds). Input: graph G and associated vector x.

Ouput: vector y, which is maximal with respect to x and $P(G)$, and a set $A\subseteq E$, which minimizes $[x(A)+r(\bar{A}): A\subseteq E]$.

1. Set $y=0$ (note that $0\leqq x$, and $0\in P(G)$).
2. Set $A=E$.
3. For $j=1$ to $|E|$ do steps 4, 5, and 6.
4. Find $\varepsilon_j=\min[|B|-1-y(\gamma(B)): j\in\gamma(B), B\subseteq V]$.
5. If $y_j+\varepsilon_j\leqq x_j$ then set $A=A\backslash\gamma(B)$.
6. Set y_j to $\min[x_j, y_j+\varepsilon_j]$.

In steps 5 and 6, $y_j+\varepsilon_j$ can be replaced with ε_j since $y_j=0$, but the version above will facilitate a modification that will be introduced later.

Step 4 of the algorithm is solved as $\min[|B|+y(\overline{\gamma(B)}): j\in\gamma(B), B\subseteq V]$. Cunningham shows how this can be solved as an s, t network flow problem on a network with $|V|+2$ nodes and $2|V|+|E|$ edges, where the values of the y's show up both on edges incident with t, and on *interior* edges of the network, i.e., edges which are not adjacent to s or t.

Hence Cunningham solves the attack problem, for a given value b, with $|E|$ network flow computations: in the attack problem with parameter b, vector x is set to s/b, and the polymatroid greedy algorithm is used to find a set $A\subseteq E$ minimizing $[x(A)+r(\bar{A}): A\subseteq E]$. Hence Cunningham's method computes the strength of G using $|V||E|$ network flow computations. This gives a time bound for the method of $O(|V|^4|E|)$ for dense graphs, and $O(|V|^2|E|^2\log(|V|^2/|E|))$ for sparse graphs.

1.2. Background on the GT network flow algorithm. Goldberg [5] and Goldberg and Tarjan [6] introduced an $O(|V|^3)$ time[2] network flow algorithm (referred to as the GT algorithm) that differs in several interesting ways from previous methods. We won't repeat the details of their method, but we refer the reader to [6].[3] A good sketch of the method is also found in [4]. From § 2.3 on, familiarity with the contents of [6] will be essential, but readers not completely familiar with the GT method and analysis can read until § 2.3 to get the general idea of the present method for computing graph strength. For such readers, and for continuity, we introduce a few of the ideas of the GT method, and some of its terminology.

[2] For the sparse case, the time bound for the Goldberg-Tarjan method is $O(|V||E|\log(|V|^2/|E|))$.

[3] The details of the method in [6] are somewhat different than those presented in earlier conference preceedings. In this paper we follow the version discussed in [6].

The Goldberg–Tarjan method is a *preflow* method, meaning that during its execution the flow into a node may be larger than the flow out of the node. At the end of the execution, the preflow becomes a maximum flow. At the start of the algorithm, all the edges out of s are saturated (filled to capacity) and thereafter, no pushes from s are made. After those initial pushes from s, each node v will be labeled by a nonnegative integer value $d(v)$. A labeling will be called *valid* for a preflow f if $d(s)=|V|$, $d(t)=0$, and $d(v)<d(w)+1$ for every (directed) edge (v, w) in the residual graph G^f of f.[4] The node labels may change during the algorithm, but they never decrease. The *excess* at a node v, denoted $e(v)$, is the amount of flow into v minus the amount of flow out of v. A node v will be called *active* if $d(v)<\infty$, and $e(v)>0$. The algorithm terminates when there are no active nodes.

The initial facts about the Goldberg–Tarjan algorithm that we will use are: it can start with any initial preflow and associated valid node labeling, provided that all edges out of s are saturated; once the first preflow and valid labeling have been established, a preflow and a valid labeling are maintained throughout the algorithm; all node labels are bounded by $2|V|$; and, if $d(v)\geqq|V|$ in a valid labeling for preflow f, then v is disconnected from t in the residual graph G^f.

1.3. The GGT parametric flow method. Gallo, Grigoriadis, and Tarjan [4] make a very fundamental observation about the GT algorithm and its time analysis.[5] Suppose one wants to compute a sequence of maximum flows on a network G, where between each flow computation the capacity on each edge incident with s is either unchanged or increased, and all other capacities are left constant. We call this a *parametric flow problem.* As we will show, computation of graph strength can be viewed as a set of $|E|$ interleaved parametric flow problems, and there are many other applications where parametric network flow problems arise (many of them are discussed in [4], [10], and [11]). If the sequence of flow problems is of length $O(|V|)$, then, solving each flow as an independent computation, the dense time bound for all the flow computations is $O(|V|^4)$. The first observation in [4] is that this time can be reduced to $O(|V|^3)$ by using the GT algorithm, as we sketch below.

DEFINITION. Let G_k be the kth network in the parametric flow problem, and let $c_k(s, v)$ be the capacity of edge (s, v) in G_k. Let f_k denote the maximum s, t flow in G_k.

For each G_k in a parametric flow problem, no edge capacity in G_{k+1} is smaller than in G_k, so we can compute f_{k+1} in G_{k+1} by starting with f_k, rather than starting with the zero flow. When the GT flow algorithm is used, this simple idea is shown to solve a parametric flow problem of length $O(|V|)$ in the same time bound as for one flow. However, there is a technical point that needs some discussion. In a direct application of the idea above, when computing the maximum flow in G_{k+1} by the GT algorithm, we would start computing f_{k+1} by setting the flow in each edge (s, v) to $c_{k+1}(s, v)$ (saturating all edges out of s), leaving the flow in all other edges as in f_k. However, if all edges out of s in G_{k+1} are so saturated, then the existing node labels may not be valid for the resulting preflow, and if we set new valid node labels, they may become smaller, which must be avoided for the desired time analysis.

To avoid the above problems, when computing f_{k+1} from f_k, the original GT algorithm is modified so that it never pushes flow on an edge (s, v) if $d(v)\geqq|V|$.

[4] The edges of the residual graph G^f for a preflow f are as follows: if the flow $f(v, w)$ along the edge (v, w) is greater than zero, then there is a directed edge (w, v) in G^f; if the flow along edge (v, w) is less than its capacity, i.e., $f(v, w)<c(v, w)$, then there is a directed edge (v, w) in G^f.

[5] We will discuss this observation only in terms of the bound for dense graphs, but mention later the analogous sparse bounds.

However, no justification is given in [4] for the correctness of this change, i.e., that the resulting flow is maximum for G_{k+1}. We will discuss this point in detail here, as it will be needed for the correctness of the method in this paper as well.

DEFINITION. Let G^{f_k} be the residual graph for f_k and G_k, and let S_k be the set of nodes which cannot reach t via a directed path in G^{f_k}.

LEMMA 1.1. *There exists a maximum s, t flow in G_{k+1} where for any $v \in S_k$, $f_{k+1}(s, v) = f_k(s, v)$.*

Proof. It is well known that if g is some nonmaximum s, t flow in any graph G, then a maximum flow in G can be obtained by superimposing a maximum s, t flow in the residual graph G^g with the flow g. But if v cannot reach t in G^g, then in any maximum s, t flow in G^g there can be no flow through v. Hence when this maximum flow is superimposed with g, the resulting flow on edges incident with v is the same as in flow g.

Now f_k is a nonmaximum flow in G_{k+1}, and if v cannot reach t in G^{f_k}, it also cannot reach t in $G^{f_k}_{k+1}$ (the residual graph for the k flow in the $(k+1)$st graph), since the only possible edges in $G^{f_k}_{k+1}$ but not G^{f_k} are edges out of s. Then, by the observation in the first paragraph, the flow in f_{k+1} incident with such a node $v \in S_k$ can be assumed to be the same as in f_k. □

If flow is never pushed on edge (s, v) in the computation of f_{k+1}, then the effect is the same as if the capacity of edge (s, v) were set to $f_k(s, v)$ rather than $c_{k+1}(s, v)$. With this interpretation, edge (s, v) is saturated at the start of the $(k+1)$st flow computation—an initial condition of the GT algorithm. This justifies the detail in the GGT algorithm, that in computing f_{k+1} starting with the initial flow f_k, no additional flow is pushed on the edge (s, v) if $d(v) \geqq |V|$.

When the GT algorithm is used for the parametric flow problem, then at the end of the flow for G_k, the node labels are valid for f_k, and any node v with label greater than $|V|-1$ is disconnected from t in G^{f_k}. To compute f_{k+1} (in G_{k+1}) from f_k, the flow in each edge (s, v) is initially set to $c_{k+1}(s, v)$ if v can reach t in G^{f_k}; else it is left at $f_k(s, v)$ (as justified by Lemma 1.1). Flow in all other edges is initially left as in f_k. At this point, the last node labels in G_k are valid for this initial preflow in G_{k+1}. Hence, the initial conditions for starting the GT algorithm on G_{k+1} have been met, and f_{k+1} can be computed by the GT algorithm starting from this preflow and node labeling. Note that the node labels at this point are the same as at the end of the kth flow.

In the analysis of the GT algorithm for a single flow, the time is divided into time for nonsaturating pushes and all other work. A bound of $O(|V||E|)$ for all other work is shown, and its derivation depends only on the fact that the node labels never decrease and each label is bounded by $O(|V|)$. In the parametric flow problem, node labels also never decrease and are bounded by $O(|V|)$, so this bound on the work other than for nonsaturating pushes also holds for the parametric problem. Close examination of the bounding argument for nonsaturating pushes in the GT algorithm shows that for the parametric problem, each time the capacities of the edges out of s are increased, the total number of permitted nonsaturating pushes increases by no more than $O(|V|^2)$. Hence, over a sequence of $O(|V|)$ flow problems, the total time needed for the parametric flow problem is $O(|V|^3)$—the worst case bound for just a single flow computation. This method and analysis has a dramatic affect on established running times for the relevant applications.

For sparse graphs the bound obtained in [4] for $O(|V|)$ flows is $O(|V||E| \log(|V|^2/|E|))$, the same as for a single maximum flow. We should note that the model considered in [4] also permits nonincreasing changes in the capacities of edges incident with t. The paper also contains a second general model and result on

parametric flow, and many applications of these two general results. We note also that it was subsequently shown [14] that the dense-case bound of the GGT method can be obtained with a version of the Dinic flow algorithm in place of the GT algorithm.

2. Computing $\sigma(G)$ in $O(|V|^3|E|)$ time. Cunningham's method to compute the strength of a graph involves a sequence of $|V||E|$ flows, where edge capacities are changed between each flow. We would like to use the ideas in [4] to speed up Cunningham's solution. However, we do not see a way to use his original solution to achieve a running time for these $|V||E|$ flows faster than by executing each flow from scratch. The difficulties are that the changing capacities (y values) are on interior edges of the network used in step 4 of the polymatroid greedy algorithm, and that the changes are not monotonic, even for the $|V|$ flow problems associated with a fixed edge j.

Instead, we modify the polymatroid greedy algorithm of § 1.1, and also use a different network to solve step 4 of the greedy algorithm. The resulting method to compute the strength of G will then be viewed as $|E|$ interleaved parametric flow problems, each involving $|V|$ parameter changes, where the parametric capacities are only on edges incident with the source node s, and where the capacities are presented in nondecreasing order. The details of these two modifications appear in §§ 2.1 and 2.2.

With the above modifications, the $|E|$ interleaved parametric flow problems at first seem to fit the requirements of the GGT model. Unfortunately, the new network we use in step 4 of the polymatroid greedy algorithm may have $\Theta(|V|^2)$ nodes rather than $O(|V|)$ nodes of the network used in [3], and this naively increases all the required time bounds by a square. However, we will show that a modified version of the GT algorithm, when run on this larger network, runs in $O(|V|^3)$ time, and this bound also applies to a parametric flow problem of $O(|V|)$ flows. The details of the modification for a single network flow computation appear in § 2.3; its analysis appears in § 2.4; and its use for the parametric flow problem appears in § 2.5.

With these modifications to the polymatroid greedy algorithm, to the network used in step 4 of the greedy algorithm, and to the GT algorithm itself, the resulting method computes $\sigma(G)$ in $O(|V|^3|E|)$ worst case time, improving the dense time bound in [3] by a factor of $|V|$. A time bound of $O(|V||E|^2 \log(|V|^2/|E|))$ for computing the strength of a graph can be obtained by applying dynamic trees to the method in this paper. This improves the sparse case bound by a factor of $|V|$ over the sparse case bound in [3]. Note that dynamic trees need not be employed to improve the sparse case bound of [3] since the bound of $O(|V|^3|E|)$ is superior even to the sparse case bound on Cunningham's original method, $O(|V|^2|E|^2 \log(|V|^2/|E|))$, although the gap is small when G is extremely sparse. However, the $O(|V|^3|E|)$ method is considerably simpler than the dynamic trees approach used to obtain any of the above sparse bounds.

2.1. Bipartite network GB. Recall that in each execution of step 4 of the polymatroid greedy algorithm, j is a fixed edge of G. We solve step 4 of the polymatroid greedy algorithm (for a fixed edge j) on the following directed bipartite network GB with $|V|+|E|+2$ nodes. GB consists of a source node s, a sink node t, and two sets of nodes S and T. Node set S contains one node for every edge in G, and node set T contains one node for every node of G. The names for the nodes of S will be the same as the corresponding edges in G, and the names of the nodes of T will be the same as the corresponding nodes of G; this double naming should cause no confusion. A node e in S is connected to a node v in T if and only if v is an endpoint of the edge in G that e represents; each edge from S to T has infinite capacity. Node s is attached to each node e in S and edge (s, e) has infinite capacity if $e=j$; otherwise it has

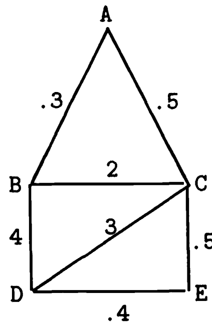
capacity $y(e)$. For each node v in T, there is a directed edge (v, t) with capacity 1 (see Fig. 1).

LEMMA 2.1. *The value of a minimum s, t cut C in GB is* $\min[|B|+y(\overline{\gamma(B)}): j \in \gamma(B), B \subseteq V]$. *Hence a minimum s, t cut in GB solves step* 4 *of the greedy algorithm.*

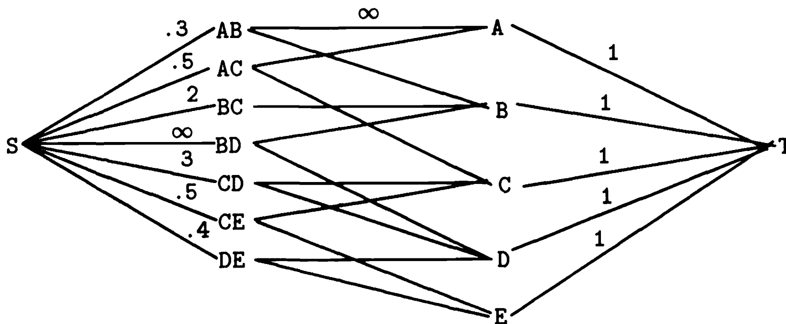
Proof. For a minimum s, t cut C in GB, define B to be the nodes $v \in T$ which are endpoints of edges (v, t) crossing C. We will show that the capacity of C is $[|B|+y(\overline{\gamma(B)})]$ and that $j \in \gamma(B)$. Since there is an s, t cut of finite capacity in GB, C cannot contain edge (s, j) nor any edges from S to T, since these edges have infinite capacity. Hence, edge j of G must be in $\gamma(B)$, else there would be an uncut s, t path in GB through node j. By definition of B, the contribution of the edges in C from T to t is exactly $|B|$. Now, by the optimality of cut C, if e is an edge of G in $\gamma(B)$, then the edge (s, e) in GB cannot cross C. Conversely, if e is not in $\gamma(B)$, then edge (s, e) must cross C, else there would be an uncut path in GB from s to t through node e (recalling that no edges from S to T can cross C). Hence the value of cut C is exactly $[|B|+y(\overline{\gamma(B)})]$ and $j \in \gamma(B)$.

Now for any subset $B \subseteq V$ such that $j \in \gamma(B)$, let C' be the set composed of all edges in GB from nodes in B of GB to t, and all edges from s to $\overline{\gamma(B)}$. It is easy to verify that C' is an s, t cut, hence its capacity cannot be less than that of C. Then, since the capacity of C' is exactly $[|B|+y(\overline{\gamma(B)})]$, it follows that the capacity of the minimum cut C is $\min[|B|+y(\overline{\gamma(B)}): j \in \gamma(B), B \subseteq V]$. □

The advantage of network GB is that, when used in Cunningham's solution to the strength problem, all edge capacities are fixed except those of edges incident with node s, as required by the GGT model.



Graph G. The numbers on the edges are the current y values.



Graph GB for $j = BD$, and the current y values.

FIG. 1

2.2. The modified greedy algorithm. We assume that step 4 of the greedy algorithm is solved by using network GB, as above. Now for a fixed edge j in G, consider the sequence of (at most) $|V|$ flow computations in GB, when edge j is selected in step 4 of the greedy algorithm. Note that each of these flows is separated by $|E|-1$ other flow computations. Let $GB(j)$ denote the *sequence* of networks used to compute these (at most) $|V|$ flows. For each fixed j, we will use the ideas of GGT to speed up the flow computations done on the sequence of networks $GB(j)$. However, these networks do not yet satisfy all the requirements of the GGT model. In $GB(j)$ only edges incident with s have changing capacity (as required by the GGT model), but other than the edge (s, j), the capacities of edges incident with s may change in a nonmonotonic way. To handle this problem, we will modify step 1 of the polymatroid greedy algorithm so that, for fixed j, the edge capacities of two successive networks in $GB(j)$ are nondecreasing.

LEMMA 2.2. *The polymatroid greedy algorithm remains correct if step* 1 *is executed only for the first attack problem, and thereafter, each execution of the greedy algorithm begins with step* 2.

Proof. First, it is known (and implicit in the correctness proof of the polymatroid greedy algorithm given in [3]) that y can be set in step 1 to any value in $P(G)$ such that $y \leqq x$. Now in the proposed modification, y is not set back to the zero vector at the start of each attack problem, but is changed only in step 6. Hence the final y vector in the solution of the ith attack problem is used as the starting y vector in the solution of the $i+1$st attack problem. So to see that the greedy algorithm still works, it is enough to note that the space $P(G)$ remains unchanged over the entire computation, and that vector x increases componentwise between each instance of the attack problem. Hence for every execution of the greedy algorithm the starting y vector is less than or equal to x (componentwise) and is in $P(G)$. Since the algorithm then changes each component of y maximally while observing the constraints that $y \leqq x$ and $y \in P(G)$, the ending y vector is a maximal $y \leqq x$ in $P(G)$. □

The modified greedy algorithm can be sped up in practice with the following observation: since the y vector is never set back to zero (as suggested above), if e is removed from A in any attack problem, it will again be removed from A in the solution to all successive attack problems. Hence step 2 need be executed only in the first attack problem, and step 3 should be modified to read: "For j in A do steps 4, 5, and 6."

Summarizing these modifications, the resulting algorithm is as follows.

MODIFIED POLYMATROID GREEDY ALGORITHM.

Input: graph G, vector x, set $A \subseteq V$, and vector y.

Ouput: vector y, which is maximal with respect to x and $P(G)$, and a set $A \subseteq E$, which minimizes $[x(A)+r(\bar{A})\colon A \subseteq E]$.

1. For j in A do steps 2, 3, and 4.
2. Find $\varepsilon_j = \min[|B|-1-y(\gamma(B))\colon j \in \gamma(B), B \subseteq V]$.
3. If $y_j + \varepsilon_j \leqq x_j$ then set $A = A \backslash \gamma(B)$.
4. Set y_j to $\min[x_j, y_j + \varepsilon_j]$.

The first time the algorithm is called, set $y=0$ and $A=E$. Thereafter, the input y and A are inherited from the previous execution of the algorithm.

2.3. Modified GT algorithm for network GB. Note that with the use of network GB, only edges incident with s have changing capacities. Further, with the modified greedy algorithm, the y vector is nondecreasing over the entire set of $|V||E|$ flows. So for each sequence of networks $GB(j)$, edge (s, j) always has infinite capacity, the

capacities of the other edges incident with s are nondecreasing, and all other capacities are constant. Hence for each j, the $|V|$ flows in $GB(j)$ are of the correct form for the GGT model. So we now have a solution to the strength problem consisting of $|V||E|$ flows, which can be considered as $|E|$ interleaved parametric flow problems of the correct form. However, we cannot yet conclude that each of these $|E|$ parametric problems can be solved in $O(|V|^3)$ amortized time, since the network that they are solved on has $|E|+|V|$ nodes, and $|E|$ could be as large as $\Theta(|V|^2)$. In order to get the $O(|V|^3)$ bound, we will modify the GT algorithm, and prove that for the networks $GB(j)$, the modified algorithm solves the flows on $GB(j)$ in $O(|V|^3)$ total time. In this and the next two sections, we assume familiarity with details and analysis of the GT algorithm [6].

We will modify the GT algorithm in a way similar to the way that the MKM algorithm was modified for bipartite flow in [13]. However, in the applications considered in [13] we have the freedom to flow from the side of the network with the fewest nodes to the side with the most nodes, making the analysis easier. In the present application, the capacities on the edges incident with s are presented in nondecreasing order, hence in order to use the parametric GT method, flow must go from the larger side S to the smaller side T.

The idea for the modified GT algorithm is very simple. Only nodes in $s \cup T$ will be given a label, and all pushes will start at a node in $s \cup T$ and will never end at a node in S. Further, all pushes will traverse a path of length two, unless the push is to t from a node $v \in T$, in which case the push goes along the single edge (v, t). Essentially, we modify the GT algorithm to look ahead one more edge than does the original algorithm.

To start the algorithm we execute the following.

For each node $e \in S$ let v be one of the two nodes in T that is adjacent to e in GB. In GB push $c(s, e)$ (the capacity of edge (s, e)) units of flow from s to v through node e. This is always possible because $c(e, v)=\infty$.

After these steps have been executed, all the edges out of s are saturated, but no nodes in S have any excess. From this point on, all pushes start from nodes in T. A push from a node v in T is a movement of flow from v to t along the single edge (v, t), or a movement of flow from v through a node e in S to a node $v' \in s \cup T$.

Immediately after the saturating pushes out of s are made, we label the nodes as follows: $d(s)=|V|$, $d(t)=0$, and for $v \in T$, $d(v)=1$; node labels are undefined for nodes in S. For a preflow f in GB, let GB^f be the residual graph for f. Validity of node labels is now defined as follows: for every node $v \in T$, $d(v)=1$ if edge (v, t) is in GB^f, and for every ordered pair of nodes (v, y) such that $v \in T$ and $y \in \{s \cup T\}$, and such that v reaches y by a path of length two in GB^f, $d(v) \leqq d(y)+1$. Further, when the label of node v is updated, it is set to the minimum of $d(w)+1$, where $w=t$ and edge (v, t) is in GB^f, or where w is in $s \cup T$, and v reaches w by a path of length two in GB^f.

A push from $v \in T$ to $v' \in T$ through node $e \in S$ is permitted only when $d(v)=d(v')+1$, and when directed edges (v, e) and (e, v') both exist in GB^f. When such a push is made, the amount pushed is the minimum of the excess at v and the residual capacity on the edge (v, e) (recall that the residual capacity of (e, v') is infinite). Similarly, a push from $v \in T$ to source s can be made through a node e in S only when $d(v)=|V|+2$ and when edges (v, e) and (e, s) both exist in GB^f. The amount pushed is the minimum of the residual capacities on those two edges, and the excess at v. A push from $v \in T$ to sink node t is possible only when $d(v)=1$ and edge (v, t) exists in GB^f.

As in the original GT algorithm, to implement the pushes from a node v in T, we keep a list I_v containing all the edges incident with v in GB. Note, however, that such lists are kept only for nodes in T. As in the original GT algorithm, when considering pushes from v, we cycle through I_v, but when (v, e) is the current edge we consider two pushes: a push from v to s through e, and also a push from v to the unique v' such that the node e is adjacent both to v and v' (i.e., such that $e=(v, v')$ in G). We can consider these two pushes from v in either order, but must update the capacity of the edge (v, e) after the first push (if possible) is done. Clearly, these pushes leave no excess at a node in S. A push from v is considered a saturating push if v still has excess after the push.

All other details of the modified GT algorithm are the same as in the original algorithm [6]. In particular, $d(v)$ is updated when a scan through I_v has been completed; a queue is kept of active nodes from which the current active node is selected (but note that the queue only contains T nodes, since no other nodes are ever active); a node v is kept as the current active node until either $d(v)$ is changed, or until v has no excess; and the algorithm terminates when no node is active.

2.4. Analysis of the modified algorithm. To prove that the modified algorithm finds a maximum flow and a minimum cut in GB we need to prove that node label validity is maintained throughout the algorithm, that a label update at the end of I_v increases d_v, that the algorithm terminates, and that the cut defined by the set of nodes that can reach t in the final GB^f separates s from t, and that this cut is saturated. These proofs are straightforward but require examination of a few more cases than in the original GT algorithm. We will examine label validity and label increase in this section; the proof of termination will follow from the discussion of complexity in the next section, and the proof of the $s-t$ cut will be given after the discussion of complexity.

LEMMA 2.3. *After the initial saturation of the edges out of s, the node labels are valid throughout the algorithm.*

Proof. Just after saturating the edges from s, the labels are clearly valid for the preflow. Thereafter, label updates and pushes to t clearly maintain validity. So the interesting cases are pushes from a node v in T either to another node v' in T or to s.

Consider a push from $v \in T$ to $v' \in T$ through node $e \in S$. Then $d(v)=d(v')+1$ and if, as a result of the push, a path from v' to v is created in the residual graph, the labels are still valid since $d(v')=d(v)-1<d(v)+1$. If edge (e, s) exists before the v to v' push, then $d(v) \leqq |V|+1$, and so $d(v') \leqq |V|=d(s)$, so if a v' to s path is created by the v to v' push, the labels are still valid. The push could also create a path from s to v, but since the algorithm begins by saturating all edges out of s, this could happen only if there previously had been a push to s through node e. Such a push could come only from v or v'. If the push to s came from v, then at that instant $d(v)$ was $|V|+1$. Now node labels never decrease and $d(s)$ is always $|V|$, so after the v to v' push the s to v path in the residual graph does not violate the validity requirement that $d(s) \leqq d(v)+1$. Alternately, suppose that the push to s came from v', so $d(v') \geqq |V|+1$ and $d(v) \geqq |V|+2$ at the time of the v to v' push. Hence, again the resulting s to v path does not violate the requirement for validity. No other new paths can be created by a v to v' push, since e is adjacent only with s and with v and v'.

Now consider a push from node $v \in T$ to node s through node e in S. This push will create a path from s to each of v and v'. The first case cannot affect validity, for $d(v)=|V|+1>d(s)$. For the second case, note that $d(v)=|V|+1 \leqq d(v')+1$, since there is an edge in the residual graph from v to e and one from e to v'. Hence $d(v') \geqq |V|$, so $d(s) \leqq d(v)+1$, and $d(s) \leqq d(v')+1$, as required for validity. The v to

s push could also create a path from v' through e to v. In that case $d(v)=|V|+1$ (since we pushed from v to s), $d(v')\leqq d(s)+1=|V|+1$ (by validity), and $d(v')\leqq |V|+2=d(v)+1$, so validity is again maintained. □

LEMMA 2.4. *The value of node label $d(v)$ strictly increases every time it is updated at the end of a scan through I_v.*

Proof. Let G^f be the residual graph at the moment that $d(v)$ is updated. The only nodes which are considered in updating $d(v)$ are those in $s\cup T$ which v reaches by a path of length two in G^f, or t if the edge (v, t) is in G^f. We first show that edge (v, t) is not in GB^f at that moment, and then show that for all other relevant nodes w, $d(v)<d(w)+1$, so that an update does increase $d(v)$.

Suppose, at the moment of update, there is an edge in GB^f from v to t. Then it must have been in G^f when (v, t) was last considered in I_v (since there are no pushes from t), but no flow was pushed on it. It follows that at the time (v, t) was considered, $d(v)<d(t)+1=1$; this is impossible, hence (v, t) is not in GB^f when $d(v)$ is updated.

Now consider node s. If, at the update of $d(v)$ there is a path from v to s through e in GB^f, then there are two cases to consider: either it was there the last time edge (v, e) was examined in the scan through I_v, or it was not there but was subsequently created by a push from a node v' in T to v through e. In the first case, $d(v)<d(s)+1$ when (v, e) was examined in I_v, and this continues to hold to the point when $d(v)$ is updated. In the second case, $d(v')=d(v)+1$ and $d(v')\leqq d(s)+1$ at the time of the v' to v push, so $d(v)\leqq d(s)$ at that point, and this continues to hold until $d(v)$ is updated.

Now consider a path from v to $v'\in T$ through e which exists at the moment of updating $d(v)$. If that path was in GB^f when (v, e) was examined in I_v, but no push was sent along it, then $d(v)<d(v')+1$, and this continues to hold until $d(v)$ is updated. If that path was not there when (v, e) was examined, then it was subsequently created by a push through e to v (creating edge (v, e)). Note that edge (e, v) is always in the residual graph since it has infinite capacity. But that push must have come from node v', since node e is adjacent only to nodes s, v, and v'. Hence at the point of that push, $d(v')=d(v)+1$, so $d(v)<d(v')+1$, and this holds until $d(v)$ is updated. The only other case is that the path from v to v' was there and was used in a push saturating the edge (v, e), but then a push from v' to v subsequently restored edge (v, e). As in the preceding case, at the time of the v' to v push, $d(v)<d(v')+1$, and this holds until $d(v)$ is updated.

Hence when $d(v)$ is updated, $d(v)<d(w)+1$, where w is any relevant node that is considered in updating $d(v)$. So $d(v)$ strictly increases when it is updated. □

LEMMA 2.5. *When executed on network GB, the modified* GT *algorithm terminates in $O(|V|^3)$ time.*

Proof. First, from the definition of valid labeling, it follows that $d(v)$ is bounded by the number of T nodes in the shortest path from v to t, and by $|V|$ plus the number of T nodes in the shortest path from v to s, in any residual graph GB^f. Since any such path must alternate between a T node and an S node, $d(v)\leqq 2|V|$ for any labeled node. This bounds the number of complete scans of any I_v by $O(|V|)$, since by Lemma 2.4 each complete scan of I_v results in an increase in $d(v)$. Now $|I_v|$ is one plus the degree of node v in G, and the cost of scanning an entry in it is still $O(1)$, even though each such scan generates two push attempts, and each push attempt examines (at most) two edges. Hence, as in the original GT algorithm and analysis, ignoring the cost for nonsaturating pushes, the cost of all other work is $O(|V||E|)$.

To bound the cost of the nonsaturating pushes, note that each one is done in $O(1)$ time, and that since $d(v)=O(|V|)$ in GB, $\sum_{v\in T} d(v)=O(|V|^2)$ at any given moment.

Note also that only the labels of T nodes change. With these observations, the analysis of the nonsaturating pushes is exactly the same as in the original GT algorithm. Hence the time needed for the nonsaturating pushes is $O(|V|^3)$, and the modified algorithm terminates in this time bound. □

LEMMA 2.6. *At termination of the modified* GT *algorithm, the preflow in GB is a maximum s, t flow.*

Proof. At termination, we partition the nodes into those that can reach t by a path in the final GB^f and those that cannot. We first show that s cannot reach t. Consider a node e in S, and let v and v' be the two nodes in T to which e is adjacent. Suppose edge (s, e) is in the final GB^f. Since (s, e) was saturated at the start of the algorithm, (s, e) must have been restored by a push to s through e from node v (say) in T. At the time of that push, $d(v)=d(s)+1=|V|+1$, and since edge (e, v') has infinite capacity, there was a path in the residual graph from v to v', so $d(v) \leqq d(v')+1$, hence $d(v') \geqq |V|$ at that moment. Node labels never decrease, so in the final GB^f either edge (s, e) is not present, or both v and v' are disconnected from t, hence s cannot reach t through e. Repeating for all e in S proves that s cannot reach t.

Clearly the algorithm has maintained a preflow throughout its execution. At termination no node has any excess, hence the final preflow f is a flow. As in the GT algorithm, it is easy to verify that the flow saturates the s, t cut above. Hence the modified algorithm finds a maximum flow and a minimum cut. □

2.5. Parametric flow in network *GB*. We now consider using the modified GT algorithm to solve the parametric flow problem on $GB(j)$, i.e., to compute $O(|V|)$ flows when the capacities on the edges out of s are nondecreasing. As discussed in § 1.3, if $d(v) \geqq |V|$ at the end of the kth flow on $GB(j)$, then the edge capacity of edge (s, v) is not set to $c_{k+1}(s, v)$, but rather to $f_k(s, v)$. Hence, after the capacities of edges out of s are set, and all edges out of s are saturated, the existing node labels are valid for the resulting preflow, and so the starting conditions for the modified GT algorithm have been met without decreasing any node labels.

THEOREM 2.1. *The modified* GT *algorithm on network GB solves the strength problem in time* $O(|V|^3|E|)$.

Proof. By the definition of the s, t cut, when edge capacities in GB are increased, additional flow from s to t will be possible as long as at least one parameterized edge crosses the cut. Suppose there are l such flow computations before no further flow increase is possible. The time needed for these l flows is divided into time for nonsaturating pushes, and all other work. Recall from the proof of Lemma 2.5 that the time for all work besides the nonsaturating pushes was bounded by $O(|V||E|)$ in the modified GT algorithm running on GB, and that this bound followed only from the fact that labels never decrease and are bounded by $O(|V|)$. In the parametric application of the algorithm, these conditions again hold, and so the time bound holds as well.

As for the time needed for the nonsaturating pushes, recall that in the analysis of the original GT algorithm, the algorithm is divided into passes, where in each pass, one nonsaturating push per node is possible. The number of passes is bounded by $\sum_v d(v)=O(|V|^2)$ so the number of nonsaturating pushes is $O(|V|^3)$. In [4] it was shown that increasing the capacities of edges out of s l times, increases the bound on the number of passes by $l \times \max[d(v)]$. By exactly the same arguments, in the modified GT algorithm each pass allows one nonsaturating push per T node, and the number of passes is again bounded by $l \times \max[d(v)]$ for l capacity changes. In GB, $|T|=|V|$, and $\max[d(v)]=O(|V|)$, so at most $O(|V|^3)$ time is added by increasing the capacities

$|V|$ times. Hence the modified GT algorithm solves the parametric problem in each $GB(j)$ in $O(|V|^3)$ time, and so the strength of a graph can be computed in $O(|V|^3|E|)$ total time. □

2.6. Dynamic trees. The GT flow algorithm for general networks can be implemented by using dynamic trees to obtain a worst case time bound of $O(|V||E|^2 \log(|V|^2/|E|))$ for a single network flow computation. This bound was also obtained in [4] for the parametric problem when the number of changes of the parameter is $O(|V|)$. Our modified GT algorithm for flow on *bipartite* graph GB can also be implemented with dynamic trees to obtain the bound of $O(|V||E|^2) \log(|V|^2/|E|)$ for computing for a single network flow, where GB has $|V|$ nodes on one side and $|E|$ nodes on the other. This bound extends directly to the parametric problem, and yields a resulting bound of $O(|V||E|^2) \log(|V|^2/|E|)$ for computing the strength of a network. As in the dense case, this bound is a factor of $|V|$ smaller than the sparse case bound presented in [3].

We will not give the details of how to use dynamic trees in our modified GT algorithm, but instead refer the reader to a more general discussion on the use of dynamic trees in the parametric flow problem on bipartite graphs [1]. That paper examines many algorithms including the modified GT algorithm presented here. Some initial details of that work are written in [16], and also stated in the final version of [4].

3. Space and other practical considerations. While the results presented in this paper are primarily of methodological importance—a significant step in the direction of obtaining a truly fast and practical solution—it is worthwhile to show that the method is presently practical for graphs of nontrivial size. To do this, we must discuss space as well as time, and show how to manage the space needs of the algorithm.

First, in terms of the time requirement, the algorithm presented in this paper should be practical for graphs of nontrivial size, even if the algorithm runs as badly as $\Omega(|V|^3|E|)$ or $\Omega(|V||E|^2 \log(|V|^2/|E|))$, i.e., as badly as the worst case upper bounds permit. To calculate the range of practicality, we assume that each step of the algorithm counted in the upper bound translates into five machine instructions, and we assume a machine with the performance characteristics of a current high-end workstation. With these assumptions, the worst case running time of the algorithm (using the best of the sparse or dense bounds) for a graph of 100 nodes and a density of 0.1 (average node degree 10) is 1.25 minutes; for 100 nodes and density 0.5 the time is 21 minutes; for 100 nodes and density 0.9 it is 38 minutes. The corresponding worst case times for the method in [3] are 2 hours, 34 hours, and 2.6 days. For 200 nodes and a density of 0.1, the time is 50 minutes, and for density 0.2 it is 3.2 hours; the times from [3] are 6.9 days and 25 days. For a graph of 500 nodes and a density of 0.01, the worst case running time is 1 hour; for a density of 0.03 it is 8.8 hours; the corresponding times from [3] are 20.8 days and 180 days.

These numbers illustrate that in terms of worst case time, the method in this paper can be considered practical, and a substantial improvement over the earlier method in [3], for a useful range of graphs—extremely dense graphs of up to 100 nodes, extremely sparse graphs of more than 500 nodes, and various combinations in between these two extremes.

3.1. Managing space. We now consider the space needs of the algorithm. For a graph with $|E|$ edges, the worst-case space needed is $\Omega(|E|^2)$. In each of the particular cases mentioned above, the main memory available on current high-end work stations

is sufficient or nearly sufficient (so it will be sufficient by the time this paper appears) to implement the algorithm. However, in the above cases we assumed a practical time limit of a few hours. If we consider larger or more dense problems which require (in worst case) much more time, then the space required may be well beyond currently available main memory space on typical machines (about 10 million words). Hence the $\Omega(|E|^2)$ space requirement would seem at first to make the method unusable for large graphs, even if we are willing to use large amounts of time. In this section we show that on real machines this space need is not a constraint.

The key observation we exploit to manage space is that the algorithm's space use is extremely local. There are $|E|$ sequences of networks, $GB(1)$ through $GB(|E|)$, that the algorithm successively returns to, and for each sequence the most recent flow and label information must be stored. So each sequence requires $\Omega(|E|)$ space. However, the algorithm returns to these sequences in a predetermined, cyclic order, completely processing the current network in a sequence $GB(i)$ before it moves on to the current network in $GB(i+1)$. Hence at any given time the algorithm needs only one network in main memory (along with the current y values which only take $O(|E|)$ space). After the current network is processed, that network will not be needed again until the current networks in all the other $|E|-1$ sequences are processed.

Suppose that the main (fast) memory of the system has room to store only $k<|E|$ networks. When a flow has been computed on the current network in $GB(i)$, that network can be sent to a disk and the current network in sequence $GB(i+k)$ $(i+k \bmod |E|)$ can be read in. In that way we always have in memory the next k needed networks. Now disk accesses and transfer is very slow but is done in parallel with the main algorithm execution, so the question is whether this use of a disk will cause a bottleneck in the running of the algorithm. That is, is the access and transfer time for a network of size $|E|$ greater than the worst-case processing time of the algorithm for k networks of size $|E|$? If so, then the use of a disk makes the worst-case time bound of the method invalid; on the other hand, if the disk transfer time is not a bottleneck (and disk space is sufficient) then the algorithm can be implemented on a real system so that computation within the worst-case time bound is assured.

The question above can be resolved by some calculations, again assuming typical numbers for currently available high-end workstations. Without giving all the details of the assumed speeds and calculations, the end result is that for all problems considered (densities up to 0.9 for graphs up to 200 nodes, up to 0.5 for graphs up to 400 nodes, and up to 0.2 for graphs up to 600 nodes) the disk transfer time from an ordinary disk is at least *three hundred* times faster than the time to process the networks in main memory; for many of the problems the transfer time is several thousand times faster.

One additional observation can be exploited to reduce space in practice. In the modified polymatroid greedy algorithm, once an edge $e=(u,v)$ is removed from A, the sequence of networks associated with e is no longer needed. Further, the edges (s,e), (e,u), (e,v), (u,t), (v,t) can be removed from all networks. The effect of removing these edges is to decrease the flow in any network by exactly two units compared to the flow before the edge removals. So all future computations can be done on smaller networks. The result is that every time an edge is removed from A, the number of networks that need to be stored decreases, as does the size of all the stored networks. Since the size of A decreases to zero during the execution of the algorithm, so does the space requirement of the algorithm. In practice this means that space use can be significantly reduced by starting with a "good" initial solution. For example, we could set the first b to be the weight of the minimum weight cut in G (which can be calculated in $O(|V|^4)$ time).

4. Additional problems. Below, we discuss two additional applications of the original GGT method, and the method of this paper. An additional application of the modified method is discussed in [7], and more complex applications are discussed in [11] and [10].

A parametric problem needing the modified GT algorithm.

Network reliability testing [12]. In a communication network G, each node v can test one incident line per day, and each line e must be tested at least once. The problem is to find a schedule to minimize the number of days to finish the tests. This problem can be solved as a parametric network flow problem in a network identical to GB, but where each edge (s, e) has a capacity of one, and each edge (v, t) has a capacity of λ. Then the problem is to find the minimum integer value of λ such that there is a flow saturating the edges out of s. Clearly, $\lambda \leqq |V|$, so these $|V|$ flows can be computed in $O(|V|^3)$ or $O(|V||E| \log (|V|^2/|E|))$ time.

In [11] the problem is generalized so that each node v can test $k(v)$ nodes per day, and each line e must be tested $t(e)$ times. By a generalization of the GGT method that allows the parameter changes to be given in *unsorted* order, we can solve this more general testing problem in time $O(|V||E| \log (|V|^2/|E|) + |E| \log (|V|^2/|E|) \log T)$ time, where $T = \sum_e t(e)$.

A parametric problem not needing the modified GT algorithm. An open question posed in [4] is to find additional applications of their parametric flow methodology. We mention here an immediate application of their methodology; in this application the flow is on a nonbipartite graph, and the original parametric GT algorithm can be used.

The rectilinear layout problem. Given m fixed points on a line, n new points must be added to that line. For each pair (i, j) of the $m+n$ points there is a given weight $w(i, j)$. The objective is to place the new points on the line to minimize $\sum_{(i,j)} [w(i, j) \times d(i, j)]$, where $d(i, j)$ is the distance between points i and j on the line. Since the original m points have fixed location, the objective function could also be expressed over the pairs consisting of two new points, and pairs of one new and one fixed point. A t-dimensional version of the problem can be defined where the points are to be placed on a t-dimensional grid, and the distance $d(i, j)$ is the t-dimensional rectilinear distance. This t-dimensional version of the problem reduces to t line problems, one in each dimension.

The layout problem can be solved by solving $m-1$ minimum cut problems on a dense network with n nodes, and hence in time $O(mn^3)$. This solution was discovered independently at least three times [15], [2], [17]. In this solution, each successive cut problem differs from the previous one in that the capacities of the edges incident with s increase and the capacities of the edges incident with t decrease and all other capacities remain fixed. Hence the previous GGT method can be applied, yielding a time bound of $O(n^3 + mn^2)$, improving the original bound of $O(mn^3)$.

In general, $m > n$ ($m \gg n$ is likely) and yet it is easy to see that over the $m-1$ cut problems the solution changes as most $n-1$ times. This suggests that the problem might be solved with only n cut computations instead of m. With this intuition, and by extending the GGT algorithm to handle changes of the parameter in *unsorted* order, we show in [11] how to solve the layout problem in $O(n^3 \log m)$ time; this is further reduced in [9] to $O(n^3 + n^2 \log m)$.

It is interesting to compare the approaches to the rectilinear layout problem of [15] and [2] in light of the results in [4]. The method in [2] differs from that in [15] primarily in that it uses the solution to a $P(k)$ problem as the starting point of the

$P(k+1)$ problem, while in [15], each such problem is solved from scratch. It was claimed in [3] that the method is faster than that in [15]; the results in [4] prove that to be true, at least by worst case measure when the GT flow algorithm is used.

Acknowledgments. I would like to thank Chip Martel for many helpful discussions on the GT parametric GT algorithms. I would also like to thank an anonymous referee for his insistence that practical issues, particularly space, be addressed.

REFERENCES

[1] R. AHUJA, J. ORLIN, C. STEIN, AND R. E. TARJAN, *Improved algorithms for bipartite network flow*, 1989, manuscript.

[2] T. Y. CHEUNG, *Mulifacility location problem with rectilinear distance by the minimum-cut approach*, ACM Trans. Math. Software, 6 (1980), pp. 549-561.

[3] W. H. CUNNINGHAM, *Optimal attack and reinforcement of a network*, J. Assoc. Comput. Mach., 32 (1985), pp. 549-561.

[4] G. GALLO, M. GRIGORIADIS, AND R. E. TARJAN, *A fast parametric network flow algorithm*, SIAM J. Comput., 18 (1989), pp. 30-55.

[5] A. GOLDBERG, *Efficient graph algorithms for sequential and parallel computers*, Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[6] A. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, J. Assoc. Comput. Mach., 4 (1988), pp. 136-146.

[7] D. GUSFIELD, *Computing the strength of a network in $O(|V|^3|E|)$ time*, Tech. Report CSE-87-2, Computer Science Division, University of California, Davis, CA, April 1987.

[8] ———, *Connectivity and edge disjoint spanning trees*, Inform. Process. Lett., 16 (1983), pp. 87-89.

[9] ———, *A faster parametric minimum cut algorithm*, Tech. Report CSE-90-11, Computer Science Division, University of California, Davis, CA, March 1990.

[10] D. GUSFIELD AND R. IRVING, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press, Cambridge, MA, 1989.

[11] D. GUSFIELD AND C. MARTEL, *A fast algorithm for the generalized parametric minimum cut problem and applications*, Algorithmica, (1990), to appear.

[12] D. GUSFIELD, C. MARTEL, AND D. FERNANDEZ-BACA, *Fast algorithms for bipartite network flow*, Tech. Report TR-356, Department of Computer Science, Yale University, New Haven, CT, 1985.

[13] ———, *Fast algorithms for bipartite network flow*, SIAM J. Comput., 16 (1987), pp. 237-251.

[14] C. MARTEL, *A comparison of phase and non-phase network algorithms*, Networks, 19 (1989), pp. 691-705.

[15] J. PICARD AND H. RATLIFF, *A cut approach to the rectilinear distance facility location problem*, Oper. Res., 26 (1978), pp. 422-433.

[16] C. STEIN, *Efficient algorithms for bipartite network flow*, manuscript, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

[17] V. A. TRUBIN, *Effective algorithm for the Weber problem with a rectangular metric*, Cybernetics, 14 (1978), pp. 874-878.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 655-668, August 1991

© 1991 Society for Industrial and Applied Mathematics

004

LOWER BOUNDS FOR ALGEBRAIC COMPUTATION TREES WITH INTEGER INPUTS*

ANDREW CHI-CHIH YAO†

Abstract. Let $W \subseteq R^n$ be *scale-invariant* and *rationally dispersed*, i.e., $\tilde{x} \in W$ implies $\lambda\tilde{x} \in W$ for all real $\lambda > 0$; and the rationals are dense in both W and $R^n - W$. It is shown that, in the algebraic computation tree model, the problem of deciding whether an input $\tilde{x} \in R^n$ with integer coordinates is a member of W has complexity $\Omega(\log_2 \hat{\beta}(W) - 2n)$, where $\hat{\beta}(W)$ is the number of connected components of W that are not of measure 0. This theorem can be used to prove tight lower bounds for the integral-constrained form of many basic problems, such as Element Distinctness, Set Disjointness, Finding Convex Hull, etc. Through further transformations, it leads to lower bounds for problems such as Integer Max Gap and Closest Pair of a simple polygon. The proof involves a nontrivial extension of the Milnor–Thom techniques for finding upper bounds on the Betti numbers of algebraic varieties.

Key words. algebraic computation trees, closest pair, integer element distinctness, lower bounds, topological approach

AMS(MOS) subject classifications. 68P10, 68Q25, 68R05

1. Introduction. The algebraic decision tree and the algebraic computation tree are two standard models for studying the complexity of computational problems which involve continuous variables. For example, given n real numbers $x_1, x_2, \cdots, x_n$, what is the intrinsic complexity of deciding whether they are all distinct? An interesting topological approach for proving lower bounds in these models was developed in [DL], [SY], [Y], [St], [B], [LW], [R], and [Se]. A very useful theorem along this line, shown by Ben-Or [B], is $C(W) = \Omega(\log \beta(W) - n)$, where $C(W)$ is the complexity of the membership problem for $W \subseteq R^n$ in the algebraic computation tree model and $\beta(W)$ is the number of connected components of W. (A similar theorem holds in the bounded-degree algebraic decision tree model.) In particular, this theorem shows that the above Element Distinctness Problem has complexity $\Omega(n \log n)$ in these models.

When the inputs are known to satisfy certain restrictions, such as the inputs being integers, the topological arguments used for proving lower bounds are no longer valid. For example, given n integers $m_1, m_2, \cdots, m_n$, no nonlinear lower bound is known for deciding whether they are all distinct. The integral-constrained case, aside from being a natural problem in itself, can also be used to provide lower bounds to the complexity of other problems. As noted by Aggarwal [AW] (see also Aggarwal, Edelsbrunner, Raghavan, and Tiwari [AERT]), an $\Omega(n \log n)$ lower bound to the integer version of some problems such as Element Distinctness will imply an $\Omega(n \log n)$ lower bound to the problem of finding the closest pair of vertices of a simple polygon. In general, it is an intriguing question in computational geometry to find out the effect of the *simplicity* assumption, i.e., when the input points are assumed to form the ordered vertices of a simple polygon (see [AGSS], [GY], and [TV]). It is hoped that the results of the present paper will help resolve this question in other instances.

For any $W \subseteq R^n$, let $\hat{\beta}(W)$ be the number of connected components of W that are not of measure 0. We will prove a general theorem which shows that, for certain sets W, a topological lower bound $C(W) = \Omega(\log \hat{\beta}(W) - 2n)$ is valid in the algebraic computation tree model, even if the inputs are restricted to be integers. In particular,

* Received by the editors August 15, 1989; accepted for publication (in revised form) November 3, 1990. This research was supported in part by National Science Foundation grant CCR-8813283.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

this can be applied to show that the Element Distinctness Problem given above has complexity $\Omega(n \log n)$ for integer inputs. The proof involves extending the Milnor–Thom techniques [M], [Th] for bounding the Betti numbers of algebraic varieties. By comparison, in previous complexity results, the Milnor–Thom theorem has always been applied in a fairly direct fashion.

We will state the general theorem in § 2, and give a list of applications of the theorem in § 3. After giving some preliminary facts in § 4, we prove the theorem in § 5. The general theorem and all the applications mentioned are also valid in the bounded-degree algebraic decision tree model, and will be discussed in § 6.

2. The main theorem. An *algebraic computation tree* models a program in which each step performs either an arithmetic operation $z \leftarrow u$ op w with op $\in \{+, -, \times, /\}$ or a branching operation $z:0$. This model was developed in [St], [Y], and [B], and we will follow the convention used in [B]. For any $D \subseteq R^n$, let $\mathcal{A}_D$ be the family of algebraic computation trees which are well defined when the inputs are restricted to D. A tree $T \in \mathcal{A}_D$ is said to *solve the membership problem for* W, if, for every input $\tilde{x} \in D$, the leaf reached by $\tilde{x}$ has output 1 if $\tilde{x} \in W$, and 0 otherwise. Let $\mathcal{A}_{D,W} \subseteq \mathcal{A}_D$ denote the family of algebraic computation trees for solving the membership problem for W. Let $C_D(W)$ denote the minimum of cost(T) for any $T \in \mathcal{A}_{D,W}$. We will be especially interested in the case $D = I^n$, where I^n is the set of all n-tuples of integers.

DEFINITION 1. For $W \subseteq R^n$, let $\beta(W)$ denote the number of connected components of W. A component is said to be *primary* if it is not of measure 0. Let $\hat{\beta}(W)$ denote the number of primary connected components of W.

DEFINITION 2. A set $W \subseteq R^n$ is said to be *scale-invariant* if $\tilde{x} \in W$ implies $\lambda\tilde{x} \in W$ for all $\lambda > 0$.

DEFINITION 3. A set $W \subseteq R^n$ is said to be *rationally dispersed* if, for every $\tilde{x} \in R^n$ and $\varepsilon > 0$, there exists a rational point $\tilde{z}$ such that $\|\tilde{z} - \tilde{x}\| < \varepsilon$ and

$$((\tilde{x} \in W) \wedge (\tilde{z} \in W)) \vee ((\tilde{x} \notin W) \wedge (\tilde{z} \notin W)) = 1.$$

Let $c_1 = 1/(1 + 2\log_2 3)$ and $c_2 = 1 + (\log_2 3)/(1 + 2\log_2 3)$.

THEOREM 1. *Let $W \subseteq R^n$ be scale-invariant and rationally dispersed. Then $C_{I^n}(W) \geqq c_1(\log_2 \beta(W) - 1) - c_2 n$.*

COROLLARY 1. *Let $W, W' \subseteq R^n$. If $W \cap W'$ is scale-invariant and rationally dispersed, then $C_{I^n}(W) \geqq c_1(\log_2 \hat{\beta}(W \cap W') - 1) - c_2 n - C_{I^n}(W')$.*

3. Applications. We give below a partial list of problems. The topological approach has given nontrivial lower bounds to the complexity of these problems when the inputs are not restricted. We will apply Theorem 1 and Corollary 1 to derive lower bounds for the integer-input formulation of these problems. In most cases, $\hat{\beta}(W)$ are clearly equal to $\beta(W)$, and we will only indicate the references where estimates of $\beta(W)$ are given.

Example 1. *Integer Element Distinctness.* Given n integers $m_1, m_2, \ldots, m_n$, decide whether they are all distinct.

In this case $W = \{\tilde{x} | \tilde{x} \in R^n, \Pi_{i \neq j}(x_i - x_j) \neq 0\}$, and $\hat{\beta}(W) = n!$ (see [SY], [B]). Theorem 1 gives $C_{I^n}(W) = \Omega(n \log n)$.

Example 2. *Integer Set Disjointness.* Given two sets of integers $A = G\{m_1, m_2, \cdots, m_n\}$ and $B = \{m'_1, m'_2, \cdots, m'_n\}$, decide whether $A \cap B = \emptyset$.

In this case $W = \{(\tilde{x}, \tilde{y}) | \tilde{x}, \tilde{y} \in R^n, \Pi_{i \neq j}(x_i - y_j) \neq 0\}$, and $\hat{\beta}(W) \geqq (n!)^2$ (see [B]). Theorem 1 gives $C_{I^n}(W) = \Omega(n \log n)$.

Example 3. *Integer Convex Hull.* Given n points with integer coordinates in the plane, decide whether the convex hull of these points has exactly n vertices.

In this case $W \in R^{2n}$ is the set of all points $(x_0, y_0, x_1, y_1, \cdots, x_{n-1}, y_{n-1})$ for which there exist permutations σ of $(0, 1, \cdots, n-1)$ such that, for all $0 \leqq i \leqq n-1$,

$$\det \begin{vmatrix} 1 & x_{\sigma_i} & y_{\sigma_i} \\ 1 & x_{\sigma_{(i+1) \bmod n}} & y_{\sigma_{(i+1) \bmod n}} \\ 1 & x_{\sigma_{(i+2) \bmod n}} & y_{\sigma_{(i+2) \bmod n}} \end{vmatrix} > 0.$$

It is known that $\hat{\beta}(W) \geqq (n-1)!$ (see [SY]). Theorem 1 gives $C_{I^n}(W) = \Omega(n \log n)$.

Example 4. *Integer Knapsack.* Given $n+1$ integers $m_1, m_2, \cdots, m_n, t$, decide whether there exists some $A \subseteq \{1, 2, \cdots, n\}$ such that $\Sigma_{i \in A} m_i = t$.

In this case, $W = \{(\tilde{x}, b) | (\tilde{x}, b) \in R^{n+1},\ \Pi_A(\Sigma_{i \in A} x_i = b) \neq 0\}$. It is known that $\hat{\beta}(W) \geqq 2^{n^2/2}$ (see [DL]). Theorem 1 gives $C_{I^n}(W) = \Omega(n^2)$.

Example 5. *Sign of Permutations.* Given n integers $m_1, m_2, \ldots, m_n$, decide whether there exists an odd permutation σ such that $m_{\sigma_1} < m_{\sigma_2} < \cdots < m_{\sigma_n}$.

In this case $W = \{\tilde{x} | \tilde{x} \in R^n, x_{\sigma_1} < x_{\sigma_2} < \cdots < x_{\sigma_n}$ for some odd permutation $\sigma\}$, and $\hat{\beta}(W) = n!/2$ (see [B]). Theorem 1 gives $C_{I^n}(W) = \Omega(n \log n)$.

Example 6. *Integer Max Gap.* Given $n+1$ integers $m_1, m_2, \cdots, m_n, t$, decide whether there exists a permutation σ such that $m_{\sigma_i} \leqq m_{\sigma_{i+1}} \leqq m_{\sigma_i} + t$ for all $1 \leqq i < n$.

In this case, $W = \{(\tilde{x}, b) | (\tilde{x}, b) \in R^{n+1}$, there exists σ such that $x_{\sigma_{i+1}} \leqq x_{\sigma_i} + b$ for all $i\}$. Let $\varepsilon_n = (4n)^{-10}$, and let W' be the set of all $(\tilde{x}, b) \in R^{n+1}$ such that

$$x_i, b > 0 \quad \text{for all } 1 \leqq i \leqq n,$$

$$\sum_{1 \leqq i \leqq n} x_i < \frac{b}{2} n(n+1) + b\varepsilon_n,$$

$$\sum_{1 \leqq i \leqq n} x_i^2 > \frac{b^2}{6} n(n+1)(2n+1) - b^2 \varepsilon_n.$$

Then $C_{I^n}(W') = O(n)$. It can also be shown that $\hat{\beta}(W \cap W') \geqq n!$ (see Appendix). An application of Corollary 1 gives $C_{I^n}(W) = \Omega(n \log n)$.

Remarks. The (noninteger) Max Gap Problem was proved in [LW] and [R] to have $\Omega(n \log n)$ complexity. The integer version was formulated by Aggarwal [AW].

Example 7. *Integer Measure.* Given $2n+1$ integers $m_1, m_2, \cdots, m_n, m_1'$, $m_2', \cdots, m_n', M$, decide whether the set $\bigcup_{1 \leqq i \leqq n} [m_i, m_i']$ has Lebesque measure M.

The Integer Max Gap Problem is reducible to this problem. Given input $m_1, m_2, \cdots, m_n, t$ for Integer Max Gap, we can compute in $O(n)$ steps $M = t + \max_i m_i - \min_j m_j$, $m_i' = m_i + t$ for $1 \leqq i \leqq n$, then solve the Measure Problem for input $m_1, m_2, \cdots, m_n, m_1', m_2', \cdots, m_n', M$. It is clear that this also gives the answer to the Integer Max Gap Problem. It follows that the Integer Measure Problem also has $\Omega(n \log n)$ complexity.

4. Notations and preparatory lemmas.

4.1. A lemma of Milnor and Thom. Let $R[x_1, x_2, \cdots, x_n]$ denote the set of polynomials in x_i with real coefficients. We write $R[\tilde{x}]$ for $R[x_1, x_2, \cdots, x_n]$ when the number of variables is clear from the context. For any $p \in R[\tilde{x}]$, let deg (p) denote the largest degree of any monomials in p. A *hypersurface in* R^n is a set H of the form $\{\tilde{x} | f(\tilde{x}) = 0,\ \tilde{x} \in R^n\}$ where $f \in R[\tilde{x}]$. We say that H is *nonsingular* if $\Sigma_{1 \leqq i \leqq n} (\partial f / \partial x_i)^2 > 0$ at all points $\tilde{x} \in H$; otherwise H is *singular.* The following result was due to Milnor [M] and Thom [Th]. Let f be a polynomial of degree $2k$ and in n variables with real coefficients. Let $H = \{\tilde{x} | f(\tilde{x}) = 0, \tilde{x} \in R^n\}$ and $W = \{\tilde{x} | f(\tilde{x}) \leqq 0,\ \tilde{x} \in R^n\}$.

LEMMA 1. *If H is nonsingular and if both H and W are compact sets, then* $\beta(W) \leqq k(2k-1)^{n-1}$.

Remarks. We refer to [M] for a proof of Lemma 1. Roughly, the sum of the Betti numbers of H is, by Morse's theory, related to the number of critical points of a certain kind of function defined on H, which can be upper bounded by an expression of the form given in the lemma. Using Alexander's duality theorem, one can relate the Betti numbers of W to the Betti numbers of H, which leads to the conclusion of the lemma. This lemma is a basic tool for deriving the Milnor–Thom theorem, which gives upper bounds to the number of components of algebraic varieties and semialgebraic sets [M], [Th]. A useful version by Ben-Or [B] states that, if W is the set of $\tilde{x} \in R^n$ satisfying $f_1(\tilde{x})=0,\ f_2(\tilde{x})=0, \cdots, f_s(\tilde{x})=0,\ g_1(\tilde{x})>0,\ g_2(\tilde{x})>0, \cdots, g_m(\tilde{x})>0$, then $\beta(W) \leqq k(2k-1)^{n+m-1}$, where $k=\max\{2, \deg(f_i), \deg(g_j)\}$. Let us call it Lemma 1′.

4.2. Notations for algebraic decision trees. An *algebraic decision tree* in R^n is a decision tree whose internal nodes contain comparisons of the form $f(x_1, x_2, \cdots, x_n):0$ where f is a polynomial with real coefficients. As the algebraic decision tree is a familiar concept, we refer to the literature (e.g., [SY]) for a detailed description.

Let $\mathcal{B}_n$ denote the family of algebraic decision trees in R^n. Let $T \in \mathcal{B}_n$. For any leaf $l \in T$, let $\xi_{T,l}$ denote the path in T from the root to l. Let $S_{T,l}$ stand for the set of input $\tilde{x} \in R^n$ that will follow the path $\xi_{T,l}$.

4.3. Basic properties of algebraic computation trees. Let $T \in \mathcal{A}_{I^n}$. An internal node of T is an *arithmetic node* if an assignment $z \leftarrow u$ op w is performed; it is a *branching node* if a branching operation $z:0$ is performed. For any leaf $l \in T$, let $\xi_{T,l}$ denote the path in T from the root to l. For $\alpha \in \{0, 1\}$, let L_α be the set of leaves l such that the output of l is α and such that $\xi_{T,l}$ traverses no edges labeled by "=." Let $S_{T,l}$ stand for the set of $\tilde{x} \in R^n$ that will trace the path $\xi_{T,l}$ if input into T.

For each arithmetic node v of T, let $r_v = p_v/q_v$ be the rational function computed at v, where p_v, $q_v \in R[\tilde{x}]$ are in their natural form (obtained from processes such as $p/q - s/t = (pt-qs)/qt$). More precisely, suppose that the assignment at v is $z \leftarrow u$ op w. For $a \in \{u, w\}$, let s_a, $t_a \in R[\tilde{x}]$ be defined as follows: if a is a variable evaluated at an ancestor node v' of v, then (s_a, t_a) is the pair of polynomials associated with v'; if $a = x_i$, then $s_a = x_i$ and $t_a = 1$; if $u = c$ is a constant, then $s_a = c$ and $t_a = 1$. We now define p_v and q_v. If op $\in \{+, -\}$, then $q_v = t_u t_w$ and $p_v = s_u t_w$ op $s_w t_u$. If op $\in \{\times\}$, then $q_v = t_u t_w$ and $p_v = s_u s_w$. If op $\in \{/\}$, then $q_v = t_u s_w$ and $p_v = s_u t_w$.

Let d_v stand for $\deg(p_v) - \deg(q_v)$, and let $d_T = 1 + \max_v \{\deg(p_v) + \deg(q_v)\}$. Let $\Delta_T = \max\{|\alpha|, 1/|\alpha| \mid \alpha \in E_T\}$, where E_T is the set of all nonzero coefficients in polynomials p_v, q_v for nodes v in T.

DEFINITION 4. Let $p(\tilde{x}) \in R[\tilde{x}]$. Define $\psi(p(\tilde{x})) = \lim_{\lambda \to 0} \lambda^s p(\tilde{x}/\lambda)$ where $s = \deg(p)$. That is, $\psi(p)$ is the sum of monomials in p of leading degree. For any rational function $r = p/q$ where $p, q \in R[\tilde{x}]$, let $\psi(r(\tilde{x})) = \psi(p(\tilde{x}))/\psi(q(\tilde{x}))$.

DEFINITION 5. A tree $T \in \mathcal{A}_{I^n}$ is said to be *normal* if, for every leaf l, the sequence of operations from the root to l is of the form

$$\begin{aligned}
&z_1 \leftarrow u_1 \,\mathrm{op}_1\, w_1,\\
&z_1 \ \mathrm{rel}_1\ 0,\\
&z_2 \leftarrow u_2 \,\mathrm{op}_2\, w_2,\\
&z_2 \ \mathrm{rel}_2\ 0,\\
&\vdots\\
&z_m \leftarrow u_m \,\mathrm{op}_m\, w_m,\\
&z_m \ \mathrm{rel}_m\ 0,
\end{aligned}$$

where $m \geqq n$; furthermore, (a) for $1 \leqq i \leqq m$, $\text{op}_i \in \{+, -, \times, /\}$, $\text{rel}_i \in \{<, =, >\}$, and u_i, w_i, and either constants or elements of $\{z_1, z_2, \cdots, z_{i-1}\} \cup \{x_1, x_2. \cdots, x_n\}$, and (b) for $1 \leqq j \leqq n$, $u_j = x_j$, $\text{op}_j = \times$, and $w_j = 1$.

DEFINITION 6. A tree $T \in \mathcal{A}_{I^n}$ is said to be *irredundant* if every node u can be traversed by some input $\tilde{x} \in I^n$ and every internal node v satisfies $\deg(p_v) + \deg(q_v) > 0$.

DEFINITION 7. A tree $T \in \mathcal{A}_{I^n}$ is said to be *regular* if T is normal and irredundant.

LEMMA 2. *If $T \in \mathcal{A}_{I^n,W}$, then there exists a regular $T' \in \mathcal{A}_{I^n,W}$ such that* $\text{cost}(T') \leqq 2 \text{cost}(T) + 2n$.

Proof. Add the sequence of instructions $z_i \leftarrow x_i \times 1$, $1 \leqq i \leqq n$, to the tree at the root. Expand each internal node into a computation node with $z_i \leftarrow u_i$ op w_i followed immediately by a branching node $z_i : 0$. Now prune away all the branches in the tree that are not traversed by any input $\tilde{x} \in I^n$. □

DEFINITION 8. For any regular $T \in \mathcal{A}_{I^n,W}$, let $\phi(T) \in \mathcal{B}_n$ denote the tree obtained from T as described below. For each branching node v in T, group the pair $(\text{parent}[v], v)$, consider it as one node $\phi(v)$ in $\phi(T)$, and associate the pair with $\psi(p_v(\tilde{x})) \cdot \psi(q_v(\tilde{x})) : 0$; each leaf l of T will be a leaf $\phi(l)$ in $\phi(T)$, with the same output. The parent-child relation in $\phi(T)$ is the natural one induced by T, with the $<$, $=$, $>$ labels on the corresponding edges.

5. Proof of Theorem 1. From the discussions in the last section, we can focus our attention on algorithms that are regular.

THEOREM 2. *Let $W \subseteq R^n$ be scale-invariant and rationally dispersed, and let $T \in \mathcal{A}_{I^n,W}$ be regular. Then there exists a set $A \subseteq R^n$ of measure 0 such that*

$$W - A \subseteq \bigcup_{l \in L_1(T)} S_{\phi(T),\phi(l)} \subseteq W,$$

and

$$R^n - W - A \subseteq \bigcup_{l \in L_0(T)} S_{\phi(T),\phi(l)} \subseteq R^n - W.$$

THEOREM 3. *Let $W \subseteq R^n$, and let $T \in \mathcal{A}_{I^n,W}$ be regular. Then, for any leaf $l \in L_0(T) \cup L_1(T)$, $\beta(S_{\phi(T),\phi(l)}) \leqq 2 \cdot 3^{n+2m}$, where $m = \text{cost}(T)/2$.*

We first show that Theorem 1 follows from Theorems 2 and 3. Let $T \in \mathcal{A}_{I^n,W}$ be any regular algebraic computation tree for the membership problem with integer inputs for W. Let $W_1, W_2, \cdots, W_t$ be the primary components of W, where $t = \hat{\beta}(W)$. Let A be the set of measure 0 given in Theorem 2. Then $W_i - A$ are nonempty for all $1 \leqq i \leqq t$. Theorems 2 and 3 imply that

$$\begin{aligned} t &\leqq |L_1(T)| 2 \cdot 3^{n+2m} \\ &\leqq 2^{m+1} \cdot 3^{n+2m}. \end{aligned}$$

It follows that

$$m \geqq c_1(\log_2 \hat{\beta}(W) - 1) - (c_2 - 1)n.$$

We have thus proved that, for any regular $T \in \mathcal{A}_{I^n,W}$,

$$\text{cost}(T) \geqq 2c_1(\log_2 \hat{\beta}(W) - 1) - 2(c_2 - 1)n.$$

Theorem 1 follows then from Lemma 2.

To prove Theorem 1, we only need to establish Theorems 2 and 3.

LEMMA 3. *Suppose $l \in L_0(T) \cup L_1(T)$ and $\tilde{x} \in S_{\phi(T),\phi(l)}$. Then $l \in L_1(T)$ if and only if $\tilde{x} \in W$.*

Proof. Let sign (c) be 1, 0, -1 depending on whether the real number c is positive, zero, or negative. As W is rationally dispersed, we can choose a rational point $\tilde{z}$ such that

$$((\tilde{x} \in W) \wedge (\tilde{z} \in W)) \vee ((\tilde{x} \notin W) \wedge (\tilde{z} \notin W)) = 1, \tag{1}$$

and for all $v \in \xi_{T,l}$,

$$\operatorname{sign}(\psi(r_v(\tilde{z}))) = \operatorname{sign}(\psi(r_v(\tilde{x}))). \tag{2}$$

Now, choose a positive integer N such that $N\tilde{z}$ is an integer point and such that, for all $v \in \xi_{T,l}$,

$$\operatorname{sign}\left(\frac{1}{N^{d_v}} r_v(N\tilde{z})\right) = \operatorname{sign}(\psi(r_v(\tilde{z}))). \tag{3}$$

From (2) and (3), we obtain that, for all $v \in \xi_{T,l}$,

$$\operatorname{sign}\left(\frac{1}{N^{d_v}} r_v(N\tilde{z})\right) = \operatorname{sign}(\psi(r_v(\tilde{x}))). \tag{4}$$

It follows from (4) that, if input into T, $N\tilde{z}$ will follow the path $\xi_{T,l}$. Since T has to give the correct output for any integer input, we conclude that l has output 1 if and only if $N\tilde{z} \in W$. Lemma 3 follows then from (1) and the fact that W is scale-invariant. □

Let A denote the set of all $\tilde{x} \in R^n$ that satisfy $\psi(p_v(\tilde{x})) \cdot \psi(q_v(\tilde{x})) = 0$ for some branching node v in T. Then A is of measure 0 as no $\psi(p_v) \cdot \psi(q_v)$ is identically zero in an irredundant T. Theorem 2 now follows from Lemma 3.

It remains to prove Theorem 3.

Remarks. This is where we need to extend the Milnor–Thom techniques. Let $z_1, z_2, \cdots, z_m$ be the variables along the path $\xi_{T,l}$ as in Definition 5. Let X be the set of all $(x_1, x_2, \cdots, x_n, z_1, z_2, \cdots, z_m) \in R^{n+m}$ satisfying $z_i = u_i \text{ op}_i\, w_i$ and $z_i \text{ rel}_i\, 0$ for all $1 \leqq i \leqq m$. Then $S_{T,l}$ is the projection of X on the first n coordinates, and thus $\beta(S_{T,l}) \leqq \beta(X)$. Since $\beta(X) \leqq 2 \cdot 3^{n+2m}$ by Lemma 1′ (see Remarks after Lemma 1), the same upper bound applies to $\beta(S_{T,l})$. However, it is $\beta(S_{\phi(T),\phi(l)})$ that needs to be estimated. We cannot apply Lemma 1′, as $S_{\phi(T),\phi(l)}$ is related to X in a less direct way.

Without loss of generality, we can assume $m > n$. Furthermore, we can assume that in the ith assignment, where $n < i \leqq m$, u_i, w_i are *nonzero* constants or elements of $\{z_1, z_2, \cdots, z_{i-1}\}$. Let $l \in L_0(T) \cup L_1(T)$. We first introduce two new sets, $V_{\varepsilon,B}$ and $V_{\lambda,\varepsilon,B,a}$, which are approximations to the set $S_{\phi(T),\phi(l)}$. Let $v_1, v_1', v_2, v_2', \cdots, v_m, v_m'$ be the sequence of nodes, alternately arithmetic and branching, on the path $\xi_{T,l}$, and let $\delta_i = 1$ or -1 depending on whether the edge out of v_i' along this path is labeled by "$>$" or "$<$." Let $p_i(\tilde{x})$, $q_i(\tilde{x})$, $r_i(\tilde{x})$ stand for $p_{v_i}(\tilde{x})$, $q_{v_i}(\tilde{x})$, $r_{v_i}(\tilde{x})$; and let $p_{i,0}(\tilde{x}), q_{i,0}(\tilde{x}), r_{i,0}(\tilde{x})$ stand for $\psi(p_{v_i}(\tilde{x})), \psi(q_{v_i}(\tilde{x})), \psi(r_{v_i}(\tilde{x}))$. We will also write d_i instead of d_{v_i}. Then

$$S_{\phi(T),\phi(l)} = \{\tilde{x} \mid \delta_i p_{i,0}(\tilde{x}) q_{i,0}(\tilde{x}) > 0, 1 \leqq i \leqq m\}. \tag{5}$$

For any ε, $B > 0$, let $V_{\varepsilon,B}$ denote the set of all $\tilde{x} \in R^n$ such that, for all $1 \leqq i \leqq m$,

$$p_i(\tilde{x})^2 \geqq \varepsilon, \qquad q_i(\tilde{x})^2 \geqq \varepsilon, \tag{6}$$

$$p_{i,0}(\tilde{x})^2 \geqq \varepsilon, \qquad q_{i,0}(\tilde{x})^2 \geqq \varepsilon, \qquad \delta_i r_{i,0}(\tilde{x}) \geqq \varepsilon, \tag{7}$$

and

$$\|\tilde{x}\|^2 + \sum_{1 \leqq i \leqq m} (r_{i,0}(\tilde{x}))^2 \leqq B. \tag{8}$$

Let λ be any positive real number. For each $1 \leqq i \leqq m$, define a polynomial $f_{i,\lambda}$ of $m+n$ variables $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m$, as described below. Suppose that the instruction at the ith computational node on $\xi_{T,l}$ is $z_i \leftarrow u$ op w. For $s \in \{u, w\}$, let

$$g_s = \begin{cases} y_j & \text{if } s = z_j,\ 1 \leqq j < i, \\ c & \text{if } s = c \text{ where } c \text{ is a constant}, \\ x_k/\lambda & \text{if } s = x_k. \end{cases}$$

Then $f_{i,\lambda}$ is $y_i - (g_u \text{ op } g_w)$ if op $\in \{+, -, \times\}$ and $y_i \cdot g_w - g_u$ if op $\in \{/\}$. For example, if the instruction is $z_5 \leftarrow z_2 - x_8$, then $f_{5,\lambda}$ is the polynomial $y_5 - (y_2 - x_8/\lambda)$.

For any positive real numbers λ, ε, B, let us consider the following polynomial in $n+2m+1$ variables $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m, u_1, u_2, \cdots, u_m, b$:

$$F_{\lambda,\varepsilon,B} = \sum_{1 \leqq i \leqq m} f_{i,\lambda}^2 + \sum_{1 \leqq i \leqq m} (\delta_i y_i \lambda^{d_i} - u_i^2 - \varepsilon)^2 + \left(\|\tilde{x}\|^2 + \sum_{1 \leqq i \leqq m} (y_i \lambda^{d_i})^2 + b^2 - B \right)^2.$$

For any $a > 0$, let $V_{\lambda,\varepsilon,B,a}$ be the set of all $\tilde{x} \in R^n$ for which there exist $(y_1, y_2, \cdots, y_m, u_1, u_2, \cdots, u_m, b)$ satisfying $F_{\lambda,\varepsilon,B}(\tilde{x}, \tilde{y}, \tilde{u}, b) \leqq a \cdot \lambda^{16(m+1)d_T}$. (Recall that $d_T = 1 + \max_v \{\deg(p_v) + \deg(q_v)\}$.)

We will now choose the values of the parameters ε, B, λ, and a. Let $W_1, W_2, \cdots, W_t$ be the t connected components of $S_{\phi(T),\phi(l)}$, where $t = \beta(S_{\phi(T),\phi(l)})$. Choose any points $\tilde{x}^{(j)} \in W_j$ such that $p_{i,0}(\tilde{x}^{(j)}) \cdot q_{i,0}(\tilde{x}^{(j)}) \neq 0$ for $1 \leqq i \leqq m$, $1 \leqq j \leqq t$. Define ε and B by

$$\varepsilon = \min \{(100m)^{-1}, p_i(\tilde{x}^{(j)})^2, q_i(\tilde{x}^{(j)})^2, \delta_i r_{i,0}(\tilde{x}^{(j)}), p_{i,0}(\tilde{x}^{(j)})^2, q_{i,0}(\tilde{x}^{(j)})^2 | i, j\},$$

$$B = 100m + \sum_{1 \leqq j \leqq t} \left(\|\tilde{x}^{(j)}\|^2 + \sum_{1 \leqq i \leqq m} (r_{i,0}(\tilde{x}^{(j)}))^2 \right).$$

Let $\Lambda = \eta^{10^{3(m+n)}}$, where $\eta = \varepsilon/(10^3 \Delta_T B)$. The following fact can be verified easily.

FACT 1. There exists a $0 < \lambda_0 < \Lambda$ such that, for all $0 < \lambda < \lambda_0$,

$$|\lambda^{d_i} r_i(\tilde{x}/\lambda) - r_{i,0}(\tilde{x})| < \varepsilon/20, \tag{9}$$

for all $\tilde{x} \in V_{\varepsilon,B}$, $1 \leqq i \leqq m$.

We further define

$$\varepsilon' = \varepsilon/5, \tag{10}$$

$$B' = 5B. \tag{11}$$

For any polynomial f, a constant c is called a *critical value* of f, if the hypersurface $\{\tilde{x} | f(\tilde{x}) - c = 0\}$ is singular. It is well known from Sard's theorem that the set of critical values of any f is of measure 0 on the real line. For each $0 < \lambda < \lambda_0$, we choose an $\frac{1}{2} \leqq a_\lambda \leqq 1$ such that $a_\lambda \cdot \lambda^{16(m+1)d_T}$ is not a critical value of $F_{\lambda,\varepsilon',B'}$.

The following two lemmas are the central properties of the approximations $V_{\varepsilon,B}$ and $V_{\lambda,\varepsilon',B',a_\lambda}$ that are needed for proving Theorem 3.

LEMMA 4. $\beta(S_{\phi(T),\phi(l)}) \leqq \beta(V_{\varepsilon,B})$.

Proof. It is easy to verify that $\tilde{x}^{(j)} \in V_{\varepsilon,B}$ for all $1 \leqq j \leqq t$. Since $V_{\varepsilon,B} \subseteq S_{\phi(T),\phi(l)}$, this proves the lemma. □

LEMMA 5. *Let* $0 < \lambda < \lambda_0$. *Then* $V_{\varepsilon,B} \subseteq V_{\lambda,\varepsilon',B',a_\lambda} \subseteq S_{\phi(T),\phi(l)}$.

For the moment, assume that Lemma 5 is true. We shall prove Theorem 3. Let $0 < \lambda < \lambda_0$. From Lemmas 4 and 5, it follows that

$$\beta(S_{\phi(T),\phi(l)}) \leqq \beta(V_{\lambda,\varepsilon',B',a_\lambda}). \tag{12}$$

However, from Lemma 1, we have

$$\beta(V') \leqq 2 \cdot 3^{n+2m},$$

where $V' = \{(\tilde{x}, \tilde{y}, \tilde{u}, b) | F_{\lambda,\varepsilon',B'} \leqq a_\lambda \cdot \lambda^{16(m+1)d_T}\}$. Since $V_{\lambda,\varepsilon',B',a_\lambda}$ is the projection of the set V' in the $\tilde{x}$ coordinates, we have

$$\beta(V_{\lambda,\varepsilon',B',a_\lambda}) \leqq 2 \cdot 3^{n+2m}. \tag{13}$$

Theorem 3 follows from (12) and (13).

It remains to prove Lemma 5.

Proof of Lemma 5. Let $0 < \lambda < \lambda_0$. We prove

$$V_{\varepsilon,B} \subseteq V_{\lambda,\varepsilon',B',a_\lambda}. \tag{14}$$

Let $\tilde{x} \in V_{\varepsilon,B}$. To prove $\tilde{x} \in V_{\lambda,\varepsilon',B',a_\lambda}$, we need to show that for some $(\tilde{y}, \tilde{u}, b) \in R^{2m+1}$, the following is true:

$$F_{\lambda,\varepsilon',B'} \leqq a_\lambda \cdot \lambda^{16(m+1)d_T}. \tag{15}$$

Define $\tilde{y} = (y_1, y_2, \cdots, y_m)$ by $y_i = r_i(\tilde{x}/\lambda)$. To prove the existence of $(\tilde{u}, b)$ satisfying (15), it is clearly sufficient to prove that

$$f_{i,\lambda}(\tilde{x}, \tilde{y}) = 0 \quad \text{for } 1 \leqq i \leqq m, \tag{16}$$

$$\delta_i y_i \lambda^{d_i} \geqq \varepsilon' \quad \text{for } 1 \leqq i \leqq m, \tag{17}$$

and

$$\|\tilde{x}\|^2 + \sum_{1 \leqq i \leqq m} (y_i \lambda^{d_i})^2 \leqq B'. \tag{18}$$

It is easy to see that (16) is true, and (17) follows from (7), (9), and (10). To prove (18), observe that, from (8) and (9), we have

$$\begin{aligned}
\|\tilde{x}\|^2 + \sum_{1 \leqq i \leqq m} (y_i \lambda^{d_i})^2 &\leqq B + \sum_{1 \leqq i \leqq m} ((y_i \lambda^{d_i})^2 - (r_{i,0}(\tilde{x}))^2) \\
&\leqq B + \sum_{1 \leqq i \leqq m} |\lambda^{d_i} r_i(\tilde{x}/\lambda) - r_{i,0}(\tilde{x})| \cdot |\lambda^{d_i} r_i(\tilde{x}/\lambda) + r_{i,0}(\tilde{x})| \\
&\leqq B + \varepsilon \sum_{1 \leqq i \leqq m} (\varepsilon + 2|r_{i,0}(\tilde{x})|) \\
&\leqq B + m\varepsilon^2 + \varepsilon \sum_{1 \leqq i \leqq m} ((r_{i,0}(\tilde{x}))^2 + 1) \\
&\leqq B + m\varepsilon^2 + m\varepsilon + B\varepsilon.
\end{aligned}$$

From the definitions of ε and B we obtain (18). This proves (14).

It remains to prove

$$V_{\lambda,\varepsilon',B',a_\lambda} \subseteq S_{\phi(T),\phi(l)}. \tag{19}$$

Let $\tilde{x} = (x_1, x_2, \cdots, x_n) \in V_{\lambda,\varepsilon',B',a_\lambda}$. We need to show

$$\delta_i p_{i,0}(\tilde{x}) q_{i,0}(\tilde{x}) > 0. \tag{20}$$

By the definition of $V_{\lambda,\varepsilon',B',a_\lambda}$, there exists a $\tilde{y} = (y_1, y_2, \cdots, y_m) \in R^m$ such that

$$|f_{i,\lambda}(\tilde{x}, \tilde{y})| \leqq \lambda^{4(m+1)d_T}, \tag{21}$$

$$\delta_i y_i \lambda^{d_i} \geqq \varepsilon'/2, \tag{22}$$

$$\|\tilde{x}\|^2 + \sum_{1 \leqq i \leqq m} (y_i \lambda^{d_i})^2 \leqq 2B', \tag{23}$$

for all $1 \leqq i \leqq m$. We will use the above inequalities to establish the next claim. Let $s_i = \deg(p_i)$ and $t_i = \deg(q_i)$.

CLAIM. *For* $1 \leqq i \leqq m$,

(A) $s_i \leqq 2^{i+2m}$, $t_i \leqq 2^{i+2m}$;

(B) $|\lambda^{t_i} q_i(\tilde{x}/\lambda)| > (\varepsilon/\Delta_T)^{4^{i+2m}}$, *and* $|q_{i,0}(\tilde{x})| > (\varepsilon/\Delta_T)^{4^{i+2m}}$;
(C) $r_i(\tilde{x}/\lambda)$ *exists, and* $|y_i - r_i(\tilde{x}/\lambda)| < \lambda^{4(m-i+1)d_T}$;
(D) $|\lambda^{s_i} p_i(\tilde{x}/\lambda)| > (\varepsilon/\Delta_T)^{4^{i+2m}}$, *and* $|p_{i,0}(\tilde{x})| > (\varepsilon/\Delta_T)^{4^{i+2m}}$;
(E) $r_{i,0}(\tilde{x})$ *exists, and* $|r_{i,0}(\tilde{x}) - \lambda^{d_i} r_i(\tilde{x}/\lambda)| < \varepsilon'/8$.

We first prove that the Claim implies (20), and hence (19). By Claims (C) and (E) we have

$$|\delta_i r_{i,0}(\tilde{x}) - \delta_i \lambda^{d_i} y_i| < \lambda^{(4(m-i)+3)d_T} + \varepsilon'/8$$
$$< \varepsilon'/4.$$

Together with (22), we then have

$$\delta_i r_{i,0}(\tilde{x}) > \varepsilon'/2 - \varepsilon'/4$$
$$> 0,$$

which is (20).

To complete the proof of Lemma 5, it remains to prove the Claim. For notational convenience, we introduce polynomials $p_i, q_i \in R[\tilde{x}]$ for negative i, and let $s_i = \deg(p_i)$, $t_i = \deg(q_i)$, $d_i = s_i - t_i$, as in the case of positive i. Let $\mu_0, \mu_1, \cdots, \mu_{m'}$ be the set of nonzero constants used in the assignments along the path $\xi_{T,l}$. Clearly, $m' < 2m$. For $0 \leqq j \leqq m'$, define $p_{-j} = \mu_j$, $q_{-j} = 1$; then $s_{-j} = t_{-j} = d_{-j} = 0$. For each $0 \leqq j \leqq m$, define the constant $y_{-j} = \mu_j$ and the rational function $r_{-j} = p_{-j}/q_{-j}$. We will now prove Claims (A)–(E) for all $-m' \leqq i \leqq m$.

It is obvious that Claims (A)–(E) are true when $-m' \leqq i \leqq 0$. Suppose $1 \leqq i \leqq n$. It follows easily from the definitions that $p_i(\tilde{x}) = p_{i,0}(\tilde{x}) = x_i$, and $q_i(\tilde{x}) = q_{i,0}(\tilde{x}) = 1$. Thus, $s_i = d_i = 1$, $t_i = 0$, and $r_i(\tilde{x}/\lambda) = x_i/\lambda$, $r_{i,0}(\tilde{x}) = x_i$ exist. Now, by (21), we have

$$|y_i - x_i/\lambda| \leqq \lambda^{4(m+1)d_T}, \tag{24}$$

which implies

$$|x_i| > |\lambda y_i| - \lambda^{4(m+1)d_T - 1}. \tag{25}$$

From (22) and (25), we obtain

$$|x_i| > \varepsilon'/4. \tag{26}$$

Using (24) and (26), one can readily verify Claims (A)–(E).

We now consider the case when i is in the range $n < i \leqq m$. Inductively, assume that Claims (A)–(E) are true for all smaller values of i, and we will establish the claims for i.

We need the following simple fact.

FACT 2. Let κ, τ, $\nu > 0$ and let $g \in R[\tilde{x}]$ be a polynomial in n variables with $\deg(g) = \nu$. If all the coefficients α of the monomial terms in g satisfy $|\alpha| \leqq \kappa$, then $|g(\tilde{x})| \leqq \kappa \cdot 2^{n+\nu} \cdot \tau^{\nu/2}$ for all $\tilde{x}$ with $\|\tilde{x}\|^2 \leqq \tau$.

Observe that $g(\tilde{x})$ is the sum of no more than $\binom{\nu+n-1}{n-1}$ monomial terms, where each term has an absolute value less than or equal to $\kappa \cdot \|\tilde{x}\|^\nu$. Fact 2 follows from the inequalities $\binom{\nu+n-1}{n-1} \leqq 2^{n+\nu}$ and $\|\tilde{x}\|^2 \leqq \tau$.

By definition, the ith assignment is $z_i \leftarrow u_i \text{ op}_i\, w_i$. It is easy to verify that $f_{i,\lambda}$, p_i, q_i have the following form for some $-m' \leqq j, k < i$.

Case 1. If $\text{op}_i \in \{+, -\}$, then $f_{i,\lambda}$ is given by $y_i - (y_j \text{ op}_i\, y_k)$, and

$$q_i = q_j q_k, \qquad p_i = p_j q_k \text{ op}_i\, q_j p_k,$$

Case 2. If $\text{op}_i \in \{\times\}$, then $f_{i,\lambda}$ is given by $y_i - y_j y_k$, and

$$q_i = q_j q_k, \qquad p_i = p_j p_k.$$

Case 3. If $\text{op}_i \in \{/\}$, then $f_{i,\lambda}$ is given by $y_i y_k - y_j$, and

$$q_i = q_j p_k, \qquad p_i = p_j q_k.$$

To prove Claim (A) for i, observe that p_j, q_j have degree at most 2^{j+2m} and p_k, q_k have degree at most 2^{k+2m} by inductive hypothesis (A). It follows that $\deg(p_i)$ and $\deg(q_i)$ are no greater than $2^{j+2m}+2^{k+2m}$, which is at most 2^{i+2m}. This proves Claim (A) for i.

Note that q_i is either q_jq_k or q_jp_k. Thus, using induction hypotheses (B) and (D) for j, k, we have

$$|\lambda^{t_i}q_i(\tilde{x}/\lambda)|>(\varepsilon/\Delta_T)^{4^{j+2m}}\cdot(\varepsilon/\Delta_T)^{4^{k+2m}} \\ >(\varepsilon/\Delta_T)^{2\cdot 4^{i-1+2m}}. \tag{27}$$

But, by (23), Fact 2, and the bound $t_i\leqq 2^{i+2m}$ just established, we have

$$\begin{aligned}|\lambda^{t_i}q_i(\tilde{x}/\lambda)-q_{i,0}(\tilde{x})|&\leqq\lambda\Delta_T\cdot 2^{n+t_i}(2B')^{t_i/2}\\ &\leqq\lambda\Delta_T\cdot(4B')^{2^{i+2m}}\\ &<\Lambda\Delta_T\cdot(20B)^{2^{i+2m}}.\end{aligned} \tag{28}$$

It follows from (27), (28), and the definition of Λ that

$$|q_{i,0}(\tilde{x})|>(\varepsilon/\Delta_T)^{3\cdot 4^{i-1+2m}}. \tag{29}$$

By (27) and (29), we have proved Claim (B) for i.

By (27) and (29), $r_i(\tilde{x}/\lambda)$ and $r_{i,0}(\tilde{x})$ exist. By induction hypothesis (C), $r_j(\tilde{x}/\lambda)$, $r_k(\tilde{x}/\lambda)$ exist and satisfy

$$|y_j-r_j(\tilde{x}/\lambda)|<\lambda^{4(m-j+1)d_T}, \tag{30}$$

and

$$|y_k-r_k(\tilde{x}/\lambda)|<\lambda^{4(m-k+1)d_T}. \tag{31}$$

We now prove Claim (C) for i, i.e., we will establish

$$|y_i-r_i(\tilde{x}/\lambda)|<\lambda^{4(m-i+1)d_T}. \tag{32}$$

There are three cases.

Case 1. $\mathrm{op}_i\in\{+,-\}$.

It follows from (21) that

$$|y_i-(y_j\ \mathrm{op}_i\ y_k)|\leqq\lambda^{4(m+1)d_T}. \tag{33}$$

Using (30), (31), and (33), we have

$$\begin{aligned}|y_i-r_i(\tilde{x}/\lambda)|&\leqq|y_i-(y_j\ \mathrm{op}_i\ y_k)|+|(y_j\ \mathrm{op}_i\ y_k)-(r_j(\tilde{x}/\lambda)\ \mathrm{op}_i\ r_k(\tilde{x}/\lambda))|\\ &\qquad+|(r_j(\tilde{x}/\lambda)\ \mathrm{op}_i\ r_k(\tilde{x}/\lambda))-r_i(\tilde{x}/\lambda)|\\ &\leqq\lambda^{4(m+1)d_T}+\lambda^{4(m-j+1)d_T}+\lambda^{4(m-k+1)d_T}\\ &\leqq 3\cdot\lambda^{4(m-i+2)d_T}\\ &\leqq\lambda^{4(m-i+1)d_T}.\end{aligned}$$

Case 2. $\mathrm{op}_i\in\{\times\}$.

It follows from (21) that

$$|y_i-y_jy_k|\leqq\lambda^{4(m+1)d_T}. \tag{34}$$

Using (30), (31), and (34), we have

$$\begin{aligned}|y_i-r_i(\tilde{x}/\lambda)|&\leqq|y_i-y_jy_k|+|y_jy_k-r_j(\tilde{x}/\lambda)r_k(\tilde{x}/\lambda)|+|(r_j(\tilde{x}/\lambda)r_k(\tilde{x}/\lambda))-r_i(\tilde{x}/\lambda)|\\ &\leqq\lambda^{4(m+1)d_T}+|y_j-r_j(\tilde{x}/\lambda)|\cdot|y_k|+|r_j(\tilde{x}/\lambda)|\cdot|y_k-r_k(\tilde{x}/\lambda)|\\ &\leqq\lambda^{4(m+1)d_T}+\lambda^{4(m-j+1)d_T}|y_k|+(|y_j|+\lambda^{4(m-j+1)d_T})\lambda^{4(m-k+1)d_T}.\end{aligned} \tag{35}$$

Now, by (23), we have

$$|y_j|,|y_k|\leqq\lambda^{-d_T}\sqrt{2B'}. \tag{36}$$

From (35) and (36), we obtain

$$|y_i - r_i(\tilde{x}/\lambda)| \leqq 4\sqrt{2B'}\,\lambda^{(4m-4i+7)d_T} \leqq \lambda^{4(m-i+1)d_T}.$$

Case 3. $\mathrm{op}_i \in \{/\}$.

It follows from (21) that

$$|y_k y_i - y_j| \leqq \lambda^{4(m+1)d_T}. \tag{37}$$

Using (30), (31), and (37), we obtain

$$\begin{aligned}|y_k|\cdot|y_i - r_i(\tilde{x}/\lambda)| &\leqq |y_k y_i - y_j| + |y_j - r_j(\tilde{x}/\lambda)| \\ &\quad + |(r_j(\tilde{x}/\lambda) - r_k(\tilde{x}/\lambda))r_i(\tilde{x}/\lambda)| + |y_k - r_k(\tilde{x}/\lambda)|\cdot|r_i(\tilde{x}/\lambda)| \\ &\leqq \lambda^{4(m+1)d_T} + \lambda^{4(m-j+1)d_T} + \lambda^{4(m-k+1)d_T}|r_i(\tilde{x}/\lambda)| \\ &\leqq \lambda^{4(m-i+2)d_T}(2 + |r_i(\tilde{x}/\lambda)|).\end{aligned} \tag{38}$$

Now, by (22), we have

$$|y_k| \geqq \varepsilon'\lambda^{d_T}/2. \tag{39}$$

From (38) and (39), we have

$$\begin{aligned}|y_i - r_i(\tilde{x}/\lambda)| &\leqq (2/\varepsilon')\lambda^{-d_T}\lambda^{4(m-i+2)}(2 + |r_i(\tilde{x}/\lambda)|) \\ &\leqq \lambda^{(4m-4i+6)d_T}(1 + |r_i(\tilde{x}/\lambda)|).\end{aligned} \tag{40}$$

This implies

$$|r_i(\tilde{x}/\lambda)| \leqq |y_i| + \lambda^{(4m-4i+6)d_T}(1 + |r_i(\tilde{x}/\lambda)|). \tag{41}$$

It follows from (23) and (41) that

$$\begin{aligned}|r_i(\tilde{x}/\lambda)| &\leqq \frac{|y_i| + \lambda^{(4m-4i+6)d_T}}{1 - \lambda^{(4m-4i+6)d_T}} \\ &\leqq 2(|y_i| + \lambda^{6d_T}) \\ &\leqq 2(\sqrt{2B'}\,\lambda^{-d_i} + \lambda^{6d_T}) \\ &\leqq 8\sqrt{B}\,\lambda^{-d_T}.\end{aligned} \tag{42}$$

From (40) and (42), we obtain finally

$$|y_i - r_i(\tilde{x}/\lambda)| \leqq \lambda^{(4m-4i+6)d_T}\cdot 9\sqrt{B}\,\lambda^{-d_T} \leqq \lambda^{4(m-i+1)d_T}.$$

Thus, (32) is true in all cases. We have established Claim (C) for i.

We next prove Claim (D) for i. Using (22) and (32), we obtain

$$\begin{aligned}|\lambda^{s_i}p_i(\tilde{x}/\lambda)/(\lambda^{t_i}q_i(\tilde{x}/\lambda))| &= |\lambda^{d_i}r_i(\tilde{x}/\lambda)| \\ &\geqq |\lambda^{d_i}y_i| - |\lambda^{d_i}y_i - \lambda^{d_i}r_i(\tilde{x}/\lambda)| \\ &> \varepsilon'/2 - \lambda^{4(m-i+1)d_T} \\ &> \varepsilon'/4.\end{aligned} \tag{43}$$

It follows from (27) and (43) that

$$\begin{aligned}|\lambda^{s_i}p_i(\tilde{x}/\lambda)| &> (\varepsilon'/4)\cdot(\varepsilon'/\Delta_T)^{2\cdot 4^{i-1+2m}} \\ &> (\varepsilon'/\Delta_T)^{3\cdot 4^{i-1+2m}}.\end{aligned} \tag{44}$$

This proves the first inequality in Claim (D).

To establish the other inequality in Claim (D), we observe that (23) and Fact 2, together with the established inequality $s_i \leqq 2^{i+2m}$ in Claim (A), imply

$$(45)\qquad \begin{aligned}|\lambda^{s_i} p_i(\tilde{x}/\lambda) - p_{i,0}(\tilde{x})| &\leqq \lambda \Delta_T \cdot 2^{n+s_i}(2B')^{s_i/2} \\ &\leqq \Lambda \Delta_T (20B)^{2^{i+2m}}\end{aligned}$$

It follows from (44) and (45) that

$$(46)\qquad \begin{aligned}|p_{i,0}(\tilde{x})| &> (\varepsilon/\Delta_T)^{3\cdot 4^{i-1+2m}} - \Lambda \Delta_T (20B)^{2^{i+2m}} \\ &> (\varepsilon/\Delta_T)^{4^{i+2m}}.\end{aligned}$$

This finishes the proof of Claim (D) for i.

Finally, we turn to the proof of Claim (E) for i. Note that

$$\begin{aligned}|r_{i,0}(\tilde{x}) - \lambda^{d_i} r_i(\tilde{x}/\lambda)| &= \left| \frac{p_{i,0}(\tilde{x})}{q_{i,0}(\tilde{x})} - \frac{\lambda^{s_i} p_i(\tilde{x}/\lambda)}{\lambda^{t_i} q_i(\tilde{x}/\lambda)} \right| \\ &= \frac{|p_{i,0}(\tilde{x}) \lambda^{t_i} q_i(\tilde{x}/\lambda) - q_{i,0}(\tilde{x}) \lambda^{s_i} p_i(\tilde{x}/\lambda)|}{|q_{i,0}(\tilde{x})| \cdot |\lambda^{t_i} q_i(\tilde{x}/\lambda)|} \\ &\leqq \frac{|p_{i,0}(\tilde{x})| \cdot |\lambda^{t_i} q_i(\tilde{x}/\lambda) - q_{i,0}(\tilde{x})| - |q_{i,0}(\tilde{x})| \cdot |\lambda^{s_i} p_i(\tilde{x}/\lambda) - p_{i,0}(\tilde{x})|}{|q_{i,0}(\tilde{x})| \cdot |\lambda^{t_i} q_i(\tilde{x}/\lambda)|}.\end{aligned}$$

Using (28) and (45), we then obtain

$$(47)\qquad |r_{i,0}(\tilde{x}) - \lambda^{d_i} r_i(\tilde{x}/\lambda)| < \Lambda \Delta_T (20B)^{2^{i+2m}} \frac{|p_{i,0}(\tilde{x})| + |q_{i,0}(\tilde{x})|}{|q_{i,0}(\tilde{x})| \cdot |\lambda^{t_i} q_i(\tilde{x}/\lambda)|}.$$

From (27), (29), and (47), we have

$$(48)\qquad |r_{i,0}(\tilde{x}) - \lambda^{d_i} r_i(\tilde{x}/\lambda)| < \Lambda \Delta_T (20B)^{2^{i+2m}} \cdot (\Delta_T/\varepsilon)^{5\cdot 4^{i-1+2m}} (|p_{i,0}(\tilde{x})| + |q_{i,0}(\tilde{x})|).$$

Now, Fact 2, (23), and the inequalities s_i, $t_i \leqq 2^{i+2m}$ established in Claim (A) imply

$$(49)\qquad \begin{aligned}|p_{i,0}(\tilde{x})| + |q_{i,0}(\tilde{x})| &\leqq 2 \cdot 2^{n+2^{i+2m}} \cdot (2B')^{2^{i+2m-1}} \\ &\leqq (5B')^{2^{3m}}.\end{aligned}$$

It follows from (48) and (49) that

$$\begin{aligned}|r_{i,0}(\tilde{x}) - \lambda^{d_i} r_i(\tilde{x}/\lambda)| &\leqq \Lambda \Delta_T (20B)^{2^{i+2m}} \cdot (\Delta_T/\varepsilon)^{5\cdot 4^{i-1+2m}} \cdot (5B')^{2^{3m}} \\ &\leqq \Lambda \cdot (10B\Delta_T/\varepsilon)^{4^{3(m+1)}} \\ &< \varepsilon'/8.\end{aligned}$$

This proves Claim (E) for i.

This completes the inductive proof of the Claim, and establishes Lemma 5. □

This also completes the proof of Theorem 1. Corollary 1 follows immediately from Theorem 1.

6. Bounded-degree decision trees. Theorem 1 and its corollary are valid for the bounded-degree algebraic decision tree model, with $c_1 = 1/(1+\log_2(2k-1))$ and $c_2 = \log_2(2k-1)/(1+\log_2(2k-1))$, where $k \geqq 2$ is the maximum degree of test polynomials allowed. The proof has the same structure as in the algebraic computation tree case but is much simpler. Define all decision trees to be *regular*, and modify Theorem 3 to read "$\cdots \beta(S_{\phi(T),\phi(l)}) \leqq k(2k-1)^{n+m-1}$ where $m = \text{cost}(T)$." All proofs remain essentially the same, except that Theorem 3 is now trivial by Lemma 1′ (as in [B]).

We learned recently that an $\Omega(n \log n)$ lower bound to the Element Distinctness Problem in the bounded-degree algebraic decision tree model was independently obtained by Lubiw and Racs [LR].

7. Concluding remarks. In this paper we have shown how to use the topological method when the input space itself is discrete. There has been considerable interest

in topics that focus on either one of these aspects ([JMW], [MST], [Sm]) in a decision tree setting. For example, Mansour, Schieber, and Tiwari [MST] studied the complexity of finding greatest common divisors of two n-bit integers. Smale [Sm] employed a topological lower bound technique not based on $\beta(W)$ for finding approximate roots of polynomials. Much work remains to be done before we can have a thorough understanding of the interplay between these two facets.

Appendix: Proof of inequality in Example 6. We will prove the inequality $\hat{\beta}(W \cap W') \geqq n!$ in Example 6. Let Ψ be the set of all permutations of $\{1, 2, \cdots, n\}$. Let $\alpha = (100n)^{-100}$, and let $\Phi \subseteq R^{n+1}$ be the set of all points of the form $p_{\sigma,b} = (b(1-\alpha)\sigma_1, b(1-\alpha)\sigma_2, \cdots, b(1-\alpha)\sigma_n, b)$, where $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_n) \in \Psi$ and $b > 0$. Let $A_\sigma = \bigcup_{b>0} \{\tilde{x} \mid \|\tilde{x} - p_{\sigma,b}\| < \alpha b\}$. It is easy to verify that, for each $\sigma \in \Psi$, $A_\sigma \subseteq W \cap W'$ and A_σ is an open set (and hence of nonzero measure). Thus, each A_σ is contained in a primary component of $W \cap W'$. It remains to prove that distinct A_σ's are contained in distinct components. It suffices to show that, if $\tilde{a} \in A_\rho$ and $\tilde{a}' \in A_{\rho'}$ where $\rho \neq \rho'$, then they must not be in the same component of $W \cap W'$.

Suppose otherwise. Let $\tilde{a} = (a_1, a_2, \cdots, a_n, a_{n+1})$, $\tilde{a}' = (a'_1, a'_2, \cdots, a'_n, a'_{n+1})$ be in the same component P. We will show that it leads to a contradiction. Let $s < t \leqq n$ be such that $a_s < a_t$ and $a'_s > a'_t$. Clearly, there exists a point $\tilde{c} = (c_1, c_2, \cdots, c_n, b) \in P$ such that $c_s = c_t$; otherwise P would be the disjoint union of two open sets $\{(x_1, x_2, \cdots, x_{n+1}) \mid x_s < x_t\} \cap P$ and $\{(x_1, x_2, \cdots, x_{n+1}) \mid x_s > x_t\} \cap P$. We will prove that

$$\text{(A1)} \qquad \sum_{1 \leqq i \leqq n} c_i^2 < b^2\left(\frac{1}{6} n(n+1)(2n+1) - \varepsilon_n\right).$$

Clearly, (A1) contradicts the assumption that $\tilde{c} \in P \subseteq W \cap W'$.

Let σ be a permutation of $\{1, 2, \cdots, n\}$ such that $c_{\sigma_1} \leqq c_{\sigma_2} \leqq \cdots \leqq c_{\sigma_{n-1}}$ with $s = \sigma_l$ and $t = \sigma_n$ where $1 \leqq l < n$. Then $c_{\sigma_l} = c_{\sigma_n}$ and, as $\tilde{c} \in W \cap W'$, we have

$$c_i, b > 0 \quad \text{for all } 1 \leqq i \leqq n,$$

$$\sum_{1 \leqq i \leqq n} c_i < b\left(\frac{1}{2} n(n+1) + \varepsilon_n\right),$$

$$|c_{\sigma_{i+1}} - c_{\sigma_i}| \leqq b \quad \text{for all } 1 \leqq i \leqq n-2.$$

It follows that

$$\begin{aligned}
2n \sum_{1 \leqq i \leqq n} c_i^2 &= \sum_{1 \leqq i, j \leqq n} (c_i - c_j)^2 + 2\left(\sum_{1 \leqq i \leqq n} c_i\right)^2 \\
&< \sum_{1 \leqq i, j < n} (c_{\sigma_i} - c_{\sigma_j})^2 + 2 \sum_{1 \leqq j < n} (c_{\sigma_n} - c_{\sigma_j})^2 + 2b^2\left(\frac{1}{2} n(n+1) + \varepsilon_n\right)^2 \\
&\leqq \sum_{1 \leqq i, j < n} b^2(i-j)^2 + 2 \sum_{1 \leqq j < n} b^2(l-j)^2 + 2b^2\left(\left(\frac{1}{2} n(n+1)\right)^2 + 4\varepsilon_n n^2\right) \\
&\leqq b^2\left(\sum_{1 \leqq i, j < n} (i-j)^2 + 2 \sum_{1 \leqq j < n} (n-j)^2 - 2 + 2\left(\sum_{1 \leqq i \leqq n} i\right)^2 + 8\varepsilon_n n^2\right) \\
&< b^2\left(\sum_{1 \leqq i, j \leqq n} (i-j)^2 + 2\left(\sum_{1 \leqq i \leqq n} i\right)^2 - 1\right) \\
&= b^2\left(2n \sum_{1 \leqq i \leqq n} i^2 - 1\right) \\
&= 2nb^2\left(\frac{1}{6} n(n+1)(2n+1) - \frac{1}{2n}\right).
\end{aligned}$$

This immediately implies (A1).

REFERENCES

[AERT] A. Aggarwal, H. Edelsbrunner, P. Raghavan, and P. Tiwari, *Optimal time bounds for some proximity problems in the plane*, manuscript, July 1989.

[AGSS] A. Aggarwal, L. J. Guibas, J. Saxe, and P. Shor, *A linear time algorithm for computing the Voronoi diagram of a convex polygon*, in Proc. 19th ACM Symposium on Theory of Computing, New York, 1987, pp. 39–45.

[AW] A. Aggarwal and J. Wein, *Computational Geometry*, Tech. Report MIT/LCS/RSS 3, Lecture Notes for course 18.409, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[B] M. Ben-Or, *Lower bounds for algebraic computation trees*, in Proc. 15th ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 80–86.

[DL] D. Dobkin and R. J. Lipton, *A lower bound of* $\frac{1}{2}n^2$ *on linear search tree programs for the knapsack problem*, J. Comput. System Sci., 16 (1978), pp. 413–417.

[GY] R. L. Graham and F. F. Yao, *Finding the convex hull of a simple polygon*, J. Algorithms, 4 (1983), pp. 324–331.

[JMW] B. Just, F. Meyer auf der Heide, and A. Wigderson, *On computations with integer division*, in Lecture Notes in Computer Science 294, R. Cori and M. Wirsing, eds., Springer-Verlag, Berlin, New York, 1988, pp. 29–37.

[LW] D. T. Lee and Y. F. Wu, *Geometric complexity of some location problems*, Algorithmica, 1 (1986), pp. 193–211.

[LR] A. Lubiw and A. Racs, *A lower bound for the integer distinctness problem*, abstract in the French-Israel Conference on Combinatorics, Israel, November 1988.

[MST] Y. Mansour, B. Schieber, and P. Tiwari, *Lower bounds for integer greatest common divisor computations*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, White Plains, NY, 1988, pp. 54–63.

[M] J. Milnor, *On the Betti numbers of real varieties*, Proc. Amer. Math. Soc., 15 (1964), pp. 275–280.

[R] P. Ramanan, *Obtaining lower bounds using artificial components*, Inform. Process. Lett., 24 (1987), pp. 243–246.

[Se] J. Seiferas, *A variant of Ben-Or's lower bound for algebraic decision trees*, Inform. Process. Lett., 26 (1988), pp. 273–276.

[Sm] S. Smale, *On the topology of algorithms*: I, preprint, Department of Mathematics, University of California, Berkeley, CA, 1987.

[SY] J. M. Steele and A. C. Yao, *Lower bounds for algebraic decision trees*, J. Algorithms, 3 (1982), pp. 1–8.

[St] V. Strassen, *The computational complexity of continued fractions*, SIAM J. Comput., 12 (1983), pp. 1–27.

[TV] R. E. Tarjan and C. Van Wyk, *An* $O(n \log\log n)$*-time algorithm for triangulating a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143–178; Erratum, SIAM J. Comput., 17 (1988), p. 1061.

[Th] R. Thom, *Sur l'homologie des variétés algébriques réelles*, in Differential and Combinatorial Topology, S. S. Cairns, ed., Princeton University Press, Princeton, NJ, 1965.

[Y] A. C. Yao, *On parallel computation for the knapsack problem*, J. Assoc. Comput. Mach., 29 (1982), pp. 898–903.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 669-685, August 1991
© 1991 Society for Industrial and Applied Mathematics
005

MINIMIZATION OF RATIONAL WORD FUNCTIONS*

CHRISTOPHE REUTENAUER† AND MARCEL-PAUL SCHUTZENBERGER‡

Abstract. Rational functions from a free monoid into another are characterized by the finiteness of the index of some congruence naturally associated with the function. A sequential bimachine is constructed computing the function, which is completely canonical, and in some sense minimal. This generalizes the Nerode criterion and the minimal automaton of a rational language, and similar results for sequential functions.

Key words. rational function, sequential bimachine

AMS(MOS) subject classification. 68D15

1. Introduction. Sequential machines appear as a ubiquitous tool in data processing and in basic software, since they constitute the most general algorithm between words that can be executed in real time by a finite device. Their theory is one of the earliest well-developed chapters of Automata Theory [8], and their natural generalization, i.e., the rational functions from a free monoid A^* (set of input words) to another B^* (output words) plays a basic role in the study of context-free languages and compilation [1]. The present paper is a contribution to the understanding of rational functions.

Here and in the sequel, we follow Eilenberg's terminology as used in his treatise [7]. In particular, by *function* we mean a partial mapping, and we recall that a rational function α from a semigroup S into a semigroup T is a function such that its graph $\{(s, \alpha(s)) \mid s \in \operatorname{dom}(\alpha)\}$ is a *rational* subset of the product semigroup $S \times T$. This definition is not the most convenient for our present purposes, and we shall use other equivalent definitions, by means of automata and machines. In order to understand the concepts which motivate the study of these objects, we begin with an informal presentation of the topic.

Recall that a sequential automaton is a two-tape machine reading the input tape from left to right, and writing on the output tape from left to right; no left move, nor ε-move, is allowed. A sequential function is by definition a function $\alpha: A^* \to B^*$ which is realized by some sequential automaton. Sequential functions are closed under functional composition.

Strictly speaking, what we have just described are left sequential objects and one could consider right sequential ones in a symmetric way (read and write from right to left). However, the associated functions are quite different. For instance, in a fixed integer base, multiplication by a given integer can be carried out by a sequential automaton if and only if it reads from right to left, while it is the reverse that is true for the division.

This leads to a more intuitive definition of rational function as the closure under composition of left and right sequential functions. An early theorem of Elgot and Mezei on general rational relations (see [1, Chap. 4, Thm. 5.2]) shows that any rational function can be obtained by composing one left and one right sequential function. This is expressed in more compact fashion by the concept of a bimachine [12] according

* Received by the editors November 20, 1989; accepted for publication (in revised form) October 23, 1990.
† Département de Mathématique et d'Informatique, Université du Québec à Montréal, Montréal, Québec, Canada H3C 3P8.
‡ Académie des Sciences, Paris, France, and 97 rue du Ranelagh, 75016, Paris, France.

to Eilenberg's terminology [7]. A further basic property that we shall make use of is that if α is an injective rational function of its domain, its inverse α^{-1} is again a rational function. For instance, morphisms $\varphi: A^* \to B^*$ may be the simplest rational functions. They are both left and right sequential functions. Another way of stating that a morphism φ is injective is the condition that the image $\varphi(A)$ of the input alphabet is a code, and in this case the decoding function φ^{-1} has been intensively studied (see [2]).

The main result of this paper is a characterization of rational functions, which extends to functions the classical definition of recognizable languages in terms of finiteness of the index of a certain congruence (Theorem 1). As a byproduct, this shortens considerably the proof of a Hankel-like characterization of rational functions [13]. The second main result (Theorem 2) shows that it is possible to associate to a rational function α a bimachine that is completely canonical, up to the choice of a certain left congruence on A^* which must be compatible with the left *adjacency* relation of α. Among these congruences, there is one, the *syntactic congruence*, which is canonical. When α is a total function, the bimachine that we construct is minimal in the following sense: it has the minimum number of left states among all bimachines computing α and having the set of right states corresponding to the given congruence. In general, it is not true that α has a unique minimal device realizing it (see, for instance, [3] for the case of decoding functions) but our result is the first step in this direction. The existence of a canonical machine is far from being trivial because, in view of the two-sided action, there is an unbounded number of ways by which one can realize the necessary trade-off between the spaces of left and right states.

Of course, the construction of a canonical bimachine gives a decision procedure for the equivalence of two rational functions (the fact that this is decidable was already known, see [1]). One can expect that, similar to the close relation between combinatorial aspects of rational languages and algebraic properties of their syntactic monoid, there should exist connections between properties of a rational function and its canonical bimachine (see the open problems at the end of this paper).

2. Preliminary results. Recall that a subset of a monoid M is called *rational* if it may be obtained from the finite subsets of M by a finite sequence of the following three operations: union $\mathbf{K} \cup \mathbf{L}$, product $\mathbf{KL}$, star $\mathbf{K}^* = \bigcup_{n \geqq 0} \mathbf{K}^n$ = the submonoid generated by $\mathbf{K}$ (see [1], [6]).

We prefer the terminology "rational" to "regular," because the former emphasizes the analogy with the theory of rational functions of classical analysis and of rational power series in noncommuting variables.

We consider here partial functions from a finitely generated free monoid into another. If $\alpha: A^* \to B^*$ is such a function, then it is called *rational* if its graph $\# \alpha = \{(u, v) \in A^* \times B^* \mid u \in \mathrm{dom}(\alpha), v = \alpha(u)\}$ is a rational subset of the product monoid $A^* \times B^*$.

In the sequel, we identify each word w and the subset $\{w\}$. We write $\alpha(w) = \varnothing$, if w is not in the domain of α.

A more effective characterization is the following: the function α is rational if and only if there exists a matrix representation (monoid homomorphism) $\mu: A^* \to (2^{B^*})^{n \times n}$, where 2^{B^*} is the boolean semiring of subsets of B^* (with union and product), a row vector λ, and a column vector ρ of length n with entries in the same semiring, such that for any word w, one has $\alpha(w) = \lambda \mu(w) \rho$ ((see [1, Chap. 3, Prop. 7.3]); the fact that α is a function forces each entry of μ, λ, ρ to be empty or a singleton, once the unnecessary states have been removed). The latter characterization shows that a

rational function has the following property, which is called the *Hankel property*, because it concerns the Hankel matrix $(\alpha(uv))_{u,v\in A^*}$.

LEMMA 1 (Hankel property). *For any rational function α, there exists an integer n and $2n$ functions $\beta_1,\cdots,\beta_n,\gamma_1,\cdots,\gamma_n : A^*\to B^*$ such that for any words x, y in A^**

$$\alpha(xy)=\bigcup_{1\leqq i\leqq n}\beta_i(x)\gamma_i(y).$$

Here and in the sequel, we consider each word, and $\emptyset$, to be embedded in the boolean semiring 2^{B^*}, with union and product; thus the previous equation means that for each i with $x\in\operatorname{dom}(\beta_i)$, $y\in\operatorname{dom}(\gamma_i)$, one has $\alpha(xy)=\beta_i(x)\gamma_i(y)$, and that $\alpha(xy)=\emptyset$ if for no i one has $x\in\operatorname{dom}(\beta_i)$ and $y\in\operatorname{dom}(\gamma_i)$.

Proof. Let μ, λ, ρ be as in the characterization before the lemma. Then

$$\alpha(xy)=\lambda\mu(xy)\rho=\lambda\mu x\mu y\rho=\bigcup_{1\leqq i\leqq n}(\lambda\mu x)_i(\mu y\rho)_i=\bigcup\beta_i(x)\gamma_i(y).$$

To conclude, note that if $|\beta_i(x)|\geqq 2$ for some x (in case β_i is not a function), one must have $\gamma_i=\emptyset$, because α is a function; so this index i can be omitted (the case is similar if some γ_i is not a function). □

A result of Schützenberger shows that the converse also holds [13]. We shall give a new proof of it in the next section. For the moment, let us point out what this Hankel property means in the case of characteristic functions, i.e., functions whose image is contained in $\{\emptyset, 1\}$ (we denote by 1 the empty word).

LEMMA 2. *Let $\alpha : A^*\to B^*$ be the characteristic function of its domain L. The following conditions are equivalent*:

(i) *α has the Hankel property.*

(ii) *$c(L)$ is a finite union $\bigcup H_i\times K_i$, where $c(w)=\bigcup_{w=xy}(x,y)\subset A^*\times A^*$.*

(iii) *L is a rational language.*

Note that (ii) is a Hopf-algebra-like characterization of rational languages.

Proof. (i)⇒(ii): Let $H_i=\operatorname{dom}(\beta_i)$ and $K_i=\operatorname{dom}(\gamma_i)$, where β_i and γ_i satisfy $\alpha(xy)=\bigcup_{1\leqq i\leqq n}\beta_i(x)\gamma_i(y)$. Then clearly $c(L)=\bigcup H_i\times K_i$.

(ii)⇒(iii): this is evident by "Nerode's criterion": if the set $\{x^{-1}L\mid x\in A^*\}$ is finite, then L is rational, where $x^{-1}L=\{y\mid xy\in L\}$. Now, $x^{-1}L$ is the union of the K_i's for which $x\in H_i$. Hence the $x^{-1}L$ are finite in number.

(iii)⇒(i) is a particular case of Lemma 1. □

The next lemma shows the functorial properties of the Hankel property.

LEMMA 3. (i) *If α and α' satisfy the Hankel property, then so does $\alpha'\circ\alpha$.*

(ii) *If α satisfies the Hankel property, then* $\operatorname{dom}(\alpha)$ *is rational.*

(iii) *If α satisfies the Hankel property, then α^{-1} preserves rationality.*

Proof. (i) We have

$$\alpha'\circ\alpha(xy)=\alpha'\left(\bigcup_i\beta_i(x)\gamma_i(y)\right)=\bigcup_i\alpha'(\beta_i(x)\gamma_i(y))$$

$$=\bigcup_i\bigcup_{i'}\beta_{i'}(\beta_i(x))\gamma_{i'}(\gamma_i(y))=\bigcup_{i,i'}(\beta_{i'}\circ\beta_i)(x)(\gamma_{i'}\circ\gamma_i)(y).$$

(ii) In this case, the characteristic function of $\operatorname{dom}(\alpha)$ satisfies the Hankel property, so it is rational by Lemma 2.

(iii) Let L be a rational language in B^*, and let $\alpha : A^*\to B^*$ satisfy the Hankel property. Let α' be the characteristic function of L. Then by Lemma 2 and (i), $\alpha'\circ\alpha$ satisfies the Hankel property, hence by (ii), $\operatorname{dom}(\alpha'\circ\alpha)$ is rational. But $\operatorname{dom}(\alpha'\circ\alpha)=\alpha^{-1}(L)$. □

This lemma will enable us to prove the following implication: if α has the Hankel property, then α is a rational function. Proving it is much more difficult than in the case of characteristic functions (Lemma 2). It depends on a Nerode-like characterization of rational functions (the main result of § 3), and on Choffrut's theorem, which characterizes subsequential functions, and which is itself a generalization of the Ginsburg–Rose theorem on sequential functions. In order to state this theorem, define the *left distance* between two words by

$$\|u, v\| = |u| + |v| - 2|u \wedge v|,$$

where $|u|$ is the length of u and $u \wedge v$ the longest common left factor of u and v. In other words, $\|u, v\| = |s| + |t|$ where $u = ps$, $v = pt$, and $p = u \wedge v$. This can also be expressed by the equality $\|u, v\|$ = length of the reduced word (in the free group) $u^{-1}v$, or equivalently $v^{-1}u$. From this last fact, it is immediate that $\|u, v\|$ satisfies the triangular inequality. Hence, it is a distance (see also [1, Chap. 4, § 2, p. 104]).

A function $\alpha : A^* \to B^*$ will be said to be *uniformly bounded* if for any integer k, there exists an integer K such that for all $x, y \in \operatorname{dom}(\alpha)$, $\|x, y\| \leqq k \Rightarrow \|\alpha(x), \alpha(y)\| \leqq K$. The terminology stems from the fact that such a function maps each bounded subset of dom (α) into a bounded subset of B^*, in a uniform way. Thus we do not use the terminology "bounded variation" of [4].

We shall give a formal definition of *subsequential functions* in § 4, but it seems advisable to recall now the following result.

THEOREM (Choffrut [4] or [1, Chap. 4, Thm. 2.7]). *A function α is subsequential if and only if it is uniformly bounded and α^{-1} preserves rationality.*

We say that two functions $\alpha, \beta : A^* \to B^*$ are *adjacent* if

$$\sup \{\|\alpha(f), \beta(f)\|, f \in \operatorname{dom}(\alpha) \cap \operatorname{dom}(\beta)\} < \infty.$$

The next result is a decidability result, which will imply that every construction in this paper is effective.

PROPOSITION 1. *If $\alpha, \alpha' : A^* \to B^*$ are rational functions, then one can decide if they are adjacent. In this case, the function $\alpha \wedge \alpha'$ defined by: $(\alpha \wedge \alpha')(f)$ equals the longest common left factor of $\alpha(f)$ and $\alpha'(f)$ when $f \in \operatorname{dom}(\alpha) \cap \operatorname{dom}(\alpha')$, and otherwise, $(\alpha \wedge \alpha')(f) = \alpha(f) \cup \alpha'(f)$, is rational and can be computed effectively.*

Remark 1. If α_1, α_2 are rational but not adjacent, then $\alpha_1 \wedge \alpha_2$ is not rational, in general. Define them, indeed, to be the homomorphisms $\{a_1, a_2\}^* \to t^*$ such that $\alpha_i(a_i) = t$, $\alpha_i(a_j) = 1$ for $j \neq i$.

Then $(\alpha_1 \wedge \alpha_2)(f)$ is equal to $t^{n(f)}$, where $n(f) = \inf(|f|_{a_1}, |f|_{a_2})$, which implies that $\alpha_1 \wedge \alpha_2$ is not rational (indeed, the inverse image of $(t^2)^*$, by the pumping lemma for finite automata, is not rational).

We shall need the following lemma, which is an easy consequence of a theorem of Fine and Wilf (see [9, Chap. 1, Prop. 3.5]).

LEMMA 4. *Let u, v, w, u', v', w' be words such that* $\sup\{\|uv^n w, u'v'^n w'\|, n \in \mathbb{N}\} < \infty$. *Then one has*:

(1) *For some word t, either $u' = ut$ and $tv' = vt$, or $u = u't$ and $tv = v't$.*

One of the referees pointed out that the lemma easily follows from the preliminary remark that $|v| = |v'|$.

Proof of Proposition 1. (1) Without loss of generality, we may assume that α and α' have the same domain and that $\alpha(1) = \alpha'(1) = \varnothing$. Indeed, we may restrict α and α' to dom $(\alpha) \cap$ dom $(\alpha') \backslash \{1\}$ and test the adjacency of these new functions. In this case, there exist transducers T and T' for α and α', with set of states Q, Q', initial states q_0, q_0', and unique final states q_f, q_f' (see [1, Chap. 3, Thm. 7.1]).

Define the "Kronecker product" of T and T': it is the "transducer" $\bar{T}$, with set of states $\bar{Q} = Q \times Q'$, inputs in A^*, and outputs in $B^* \times B^*$; there is a path $(p, p') \xrightarrow{x/(u,u')} (q, q')$ in $\bar{T}$ if and only if there is a path $p \xrightarrow{x/u} q$ in T and $p' \xrightarrow{x/u'} q'$ in T'; moreover, all the unnecessary states of $\bar{T}$ are removed, so that all states of $\bar{T}$ are accessible and coaccessible, with initial state $\bar{q}_0 = (q_0, q_0')$ and final state $\bar{q}_f = (q_f, q_f')$.

A *simple* path is a path without repetition of states, and a *simple* circuit is a closed path with no repetition of internal states.

We show that α and α' are adjacent if and only if $\bar{T}$ satisfies the following condition:

(C) For any simple path $(q_0, q_0') \xrightarrow{x/(u,u')} (q, q')$ and any simple circuit $(q, q') \xrightarrow{y/(v,v')} (q, q')$, we have equation (1) of Lemma 4.

Clearly, if α, α' are adjacent, and with the notations of (C), there exists a path

$$(q, q') \xrightarrow{z/(w,w')} (q_f, q_f').$$

Then $\alpha(xy^nz) = uv^nw$ and $\alpha'(xy^nz) = u'v'^nw'$. As α, α' are adjacent, Lemma 4 shows that (1) holds.

Conversely, suppose that (C) holds. Then, for each long enough word m in $\operatorname{dom}(\alpha) = \operatorname{dom}(\alpha')$, there is a factorization $m = xyz$, a simple path and a simple circuit as in (C) above, and a path $(q, q') \xrightarrow{z/(w,w')} (q_f, q_f')$.

Then $\alpha(m) = uvw$, $\alpha'(m) = u'v'w'$. By (1), we have, e.g., $u' = ut$ and $tv' = vt$. Then $u'v'w' = utv'w' = uvtw'$, hence $\|\alpha(m), \alpha'(m)\| = \|uvw, uvtw'\| = \|w, tw'\| = \|uw, utw'\| = \|uw, u'w'\| = \|\alpha(xz), \alpha'(xz)\|$, which allows us to conclude by induction on the length of m.

Clearly, condition (C) is decidable, which completes the first part of the proof.

(2) We construct now a transducer for $\alpha \wedge \alpha'$, which will imply that it is a rational function. This construction is a rather classical covering construction, so we shall not be very formal.

We call a path in $\bar{T}$ *elementary* if it starts from (q_0, q_0') and if only the last vertex is allowed to appear more than once, and in this case, only twice. Hence, such a path is either a simple path, or the concatenation of a simple path with a simple circuit, as in condition (C).

Denote by $u \wedge v$ the longest common left factor of the words u and v. We construct a tree T^* having the set of elementary paths in $\bar{T}$ as a set of nodes; there is an edge from π to π' in T^* if $\pi' = \pi e$, with e an edge in $\bar{T}$. Note that π, π' correspond to paths $(q_0, q_0') \xrightarrow{x/(u,u')} (p, p')$ and $(q_0, q_0') \xrightarrow{xa/(v,v')} (q, q')$, with u (respectively, u') a left factor of v (respectively, v'); so we have an equation $v \wedge v' = (u \wedge u')s$, for some word s in B^*: then the previously created edge in T^* will be labelled by a/s.

Call an elementary path *complete* if its last state is repeated. Now, in T^*, merge the node corresponding to such a state with its first occurrence in the path: in this way, we obtain a transducer S; let β be the function computed by S.

We show that $\beta = \alpha \wedge \alpha'$. Clearly, $\beta(m) = (\alpha \wedge \alpha')(m)$ for any word m such that there is in $\bar{T}$ an elementary path $(q_0, q_0') \xrightarrow{m/\cdots} (q_f, q_f')$.

It follows that this equality is true for each short enough word m. Now, let m be such that there is a nonelementary path $(q_0, q_0') \xrightarrow{m/\cdots} (q_f, q_f')$. Then this path may be decomposed as

$$(q_0, q_0') \xrightarrow{x/(u,u')} (q, q') \xrightarrow{y/(v,v')} (q, q') \xrightarrow{z/(w,w')} (q_f, q_f'),$$

where the first two factors form an elementary path, for some factorizations $m = xyz$, $\alpha(m) = uvw$, $\alpha'(m) = u'v'w'$. Moreover, $\alpha(xz) = uw$ and $\alpha'(xz) = u'w'$.

This corresponds in S to a path

$$(q_0, q_0') \xrightarrow{x/\bar{u}} (q, q') \xrightarrow{y/\bar{v}} (q, q') \xrightarrow{z/\bar{w}} (q_f, q_f').$$

By construction, we have $\bar{u} = u \wedge u'$, $\bar{u}\bar{v} = uv \wedge u'v'$. By induction on $|m|$, we also have $(\alpha \wedge \alpha')(xz) = \beta(xz) = \bar{u}\bar{w}$. Now, condition (C) holds, so we have, e.g., $u' = ut$ and $tv' = vt$. Hence, $\bar{u}\bar{w} = uw \wedge u'w' = uw \wedge utw' = u(w \wedge tw')$. As $\bar{u} = u \wedge u' = u$, we obtain $\bar{w} = w \wedge tw'$. Moreover, $\alpha(m) \wedge \alpha'(m) = uvw \wedge u'v'w' = uvw \wedge uvtw' = uv(w \wedge tw') = uv\bar{w}$. Now, we have also $\bar{u}\bar{v} = uv \wedge u'v' = uv \wedge uvt = uv$, so that $\alpha(m) \wedge \alpha'(m) = \bar{u}\bar{v}\bar{w} = \beta(xyz) = \beta(m)$, which had to be shown. □

3. A characterization of rational functions. We give a characterization of rational functions, which has some formal analogy with the Nerode criterion for rational languages and which is related to Choffrut's theorem (see § 2).

As we consider partial functions, it will be convenient to use symbol $\varnothing$, and the distance will be extended by setting

$$\|\varnothing, \varnothing\| = 0, \qquad \|\varnothing, u\| = \|u, \varnothing\| = \infty$$

for any word u. By convention, we have $n < \infty$ for any number n and $n + \infty = \infty$. Then, the triangular inequality remains valid. Now, let α be a fixed (partial) function $A^* \to B^*$, where A, B are finite alphabets. Define a relation

$$u \sim v$$

on A^* by the condition

$$\sup\{\|\alpha(fu), \alpha(fv)\|, f \in A^*\} < \infty.$$

Note that, by the above conventions, $u \sim v$ implies that $\alpha(fu) = \varnothing$ if and only if $\alpha(fv) = \varnothing$. This implies, by the triangular inequality, that $\sim$ is transitive. Moreover, it is clearly reflexive and symmetric and it is not difficult to show that $\sim$ is left compatible, i.e., $u \sim v \Rightarrow xu \sim xv$ for any word x. Hence $\sim$ is a left congruence of A^*.

We call it the *syntatic left congruence* of α. The terminology is justified by the following observation: if α is the characteristic partial function of a language L (i.e., $\alpha(w) = 1$ if $w = L$, $= \varnothing$ if $w \notin L$), then its syntactic left congruence is the usual syntactic left congruence of L. One could, of course, also define the right syntactic congruence in a symmetric way.

The main result of this section is given in the following theorem.

THEOREM 1. *A partial function $\alpha : A^* \to B^*$ is rational if and only if its syntactic left congruence is of finite index and if $\alpha^{-1}(L)$ is rational for any rational language $L \subset B^*$.*

A consequence of this result is a new proof of the Hankel-like characterization of [13].

COROLLARY. *A partial function $\alpha : A^* \to B^*$ is rational if and only if there exists an integer n and partial functions $\beta_i, \gamma_i : A^* \to B^*$, $1 \leqq i \leqq n$, such that for any words x, y*

$$\alpha(xy) = \bigcup_{1 \leqq i \leqq n} \beta_i(x)\gamma_i(y). \tag{2}$$

Proof. We prove the theorem and its corollary at the same time by showing that α rational$\Rightarrow \alpha$ satisfies the Hankel property$\Rightarrow \sim$ of finite index and α^{-1} preserves rationality$\Rightarrow \alpha$ rational. The first implication is Lemma 1 and one-half of the second is Lemma 3. So, assuming (2), we show that the syntactic congruence $\sim$ of α is of finite index.

We show that the condition

$$\forall i, \quad 1 \leqq i \leqq n: \gamma_i(u) \neq \varnothing \quad \text{iff } \gamma_i(v) \neq \varnothing \tag{3}$$

implies $u \sim v$: this will imply that the index of $\sim$ is less than or equal to 2^n. So, let (3) be satisfied and define N to be some integer greater than the lengths of the words $\gamma_i(u)$, $\gamma_i(v) \neq \emptyset$, $1 \leqq i \leqq n$. Let f be any word; we show that $\|\alpha(fu), \alpha(fv)\| < 2N$. Indeed, if $\alpha(fu) = \emptyset$, then by (2), for any i, either $\beta_i(f) = \emptyset$ or $\gamma_i(u) = \emptyset$. By (3) we obtain: for all i, $\beta_i(f)$ or $\gamma_i(v) = \emptyset$, and again by (2), $\alpha(fv) = \emptyset$. In this case, $\|\alpha(fu), \alpha(fv)\| = 0 < 2N$. On the other hand, if $\alpha(fu) \neq \emptyset$, then there exists by (2) an i such that $\alpha(fu) = \beta_i(f)\gamma_i(u)$ and $\beta_i(f) \neq \emptyset \neq \gamma_i(u)$. Hence, by (3), we have $\gamma_i(v) \neq \emptyset$, which implies by (2) that $\alpha(fv) = \beta_i(f)\gamma_i(v)$. Hence

$$\|\alpha(fu), \alpha(fv)\| = \|\beta_i(f)\gamma_i(u), \beta_i(f)\gamma_i(v)\| = \|\gamma_i(u), \gamma_i(v)\| < 2N.$$

Finally, we have $\sup\{\|\alpha(fu), \alpha(fv)\|, f \in A^*\} < \infty$ and thus $u \sim v$.

We now show the last implication: if $\sim$ is of finite index and if α^{-1} preserves rationality, then α is a rational function.

Since $\sim$ is a left congruence of finite index on A^*, the set

$$Q = A^*/\sim$$

is a finite set with a left action $(w, q) \mapsto wq$ of A^* on Q. Consider the finite alphabet $A \times Q$ and define a length-preserving function

$$\gamma : A^* \to (A \times Q)^*$$

by

$$\gamma(a_n \cdots a_1) = (a_n, q_{n-1}) \cdots (a_2, q_1)(a_1, q_0),$$

where $a_i \in A$, q_0 is the class of 1 mod $\sim$ and where $q_i = a_i q_{i-1}$ for $i = 1, \cdots, n-1$. This function γ is clearly sequential from right to left, and hence a rational function (see [1, Chap. 4, Cor. 2.3]). Clearly, γ is injective, hence γ^{-1} is a partial function. Actually, $\gamma^{-1} = \pi | \mathrm{Im}(\gamma)$, where π is the canonical projection

$$\pi : (A \times Q)^* \to A^*.$$

Define $\beta = \alpha \circ \gamma^{-1} : (A \times Q)^* \to B^*$. We have $\alpha = \beta \circ \gamma$ since γ is a total function. We show that β is a subsequential function, hence it is rational (see [1, Chap. 4, Prop. 2.4]); this will imply that α is rational, as a product of rational functions. (See [1, Chap. 3, Thm. 4.4 and Def., § 1].)

We use Choffrut's theorem, stated in § 2. As β^{-1} clearly preserves rationality (because $\beta^{-1} = \gamma \circ \alpha^{-1}$ and γ and α^{-1} both preserve rationality), it is enough to show that β is uniformly bounded.

CLAIM. *If $FU \in \mathrm{Im}(\gamma)$ with $F \neq 1$, then the last letter of F is of the form (a, uq_0) where $u = \pi(U)$.*

This is immediate from the definition of γ.

Let k be an integer. Define K to be some integer greater than $\|\alpha(fu), \alpha(fv)\|$ for any word f and any words u, v such that $u \sim v$ and $|u| + |v| \leqq k$, and greater than $\|\beta(X), \beta(Y)\|$ for $|X| + |Y| \leqq k$ and $X, Y \in \mathrm{dom}(\beta)$.

This is possible by the definition of $\sim$ and the fact that the words u, v with $|u| + |v| \leqq k$ (respectively, the words X, Y with $|X| + |Y| \leqq k$) are finite in number.

We show that

$$\forall X, Y \in \mathrm{dom}(\beta), \qquad \|X, Y\| \leqq k \Rightarrow \|\beta(X), \beta(Y)\| \leqq K, \tag{4}$$

which will imply that β is uniformly bounded. By the definition of K, it is enough to prove (4) for $|X| + |Y| > k$.

So, let X, Y with $X, Y \in \mathrm{dom}(\beta)$, $|X|+|Y|>k$, $\|X, Y\| \leqq k$. We may write $X = FU$, $Y = FV$, where F is the longest common left factor of X and Y. Since $\|X, Y\| \leqq k$, we have $|U|+|V| \leqq k$. Since $|X|+|Y|>k$, we also have $F \neq 1$.

Let $u = \pi(U)$, $v = \pi(V)$, $f = \pi(F)$. Since $X, Y \in \mathrm{dom}(\beta)$, we have $X, Y \in \mathrm{Im}(\gamma)$; hence, by the claim, the last letter of F is $(a, uq_0) = (a, vq_0)$, and thus $uq_0 = vq_0$, which implies $u \sim v$. By the definition of β, we have $\beta(X) = \alpha(fu)$ and $\beta(Y) = \alpha(fv)$. Since $|u|+|v| = |U|+|V| \leqq k$, we have by the definition of K, $\|\alpha(fu), \alpha(fv)\| \leqq K$, i.e., $\|\beta(X), \beta(Y)\| \leqq K$, which proves (4). □

4. A canonical bimachine. We modify slightly the definition of a generalized bimachine, as given in [1] and [7]. One of the reasons for this is that we want to give an arbitrary image to the empty word under the function computed by the bimachine.

A bimachine is given by

- A finite set L of *left states*, with right action $L \times A^* \to L$, $(l, w) \mapsto lw$, and a *left initial state* l_0.
- A finite set R of *right states*, with a left action $A^* \times R \to R$, $(w, r) \mapsto wr$, and with a *right initial state* r_0.
- An *output function* $\omega : \mathbf{L} \times A \times \mathbf{R} \to B^*$.
- A *final left function* $\lambda : \mathbf{R} \to B^*$ and a *final right function* $\rho : L \to B^*$.

The output function is extended to $L \times A^* \times R$ by the formula

$$\omega(l, uv, r) = \omega(l, u, vr)\omega(lu, v, r). \tag{5}$$

In particular, $w(l, 1, r) = 1$. The function computed by the bimachine is $\alpha : A^* \to B^*$ defined by

$$\alpha(w) = \lambda(wr_0)\omega(l_0, w, r_0)\rho(l_0 w). \tag{6}$$

If $w = a_1 \cdots a_n (a_i \in A)$, this may be written more algorithmically (using (5)) as

$$\begin{aligned} \alpha(a_1 \cdots a_n) = \lambda(a_1 \cdots a_n r_0) \cdot \prod_{i=1}^{n} \alpha(l_0 a_1 \cdots a_{i-1}, a_i, a_{i+1} \cdots a_n r_0) \\ \times \rho(l_0 a_1 \cdots a_n). \end{aligned} \tag{7}$$

When R is reduced to a single element, then a bimachine is simply a subsequential transducer, as in [1] (a subsequential transducer is sometimes called a generalized sequential machine with endmarker, see [5, Thm. 2.2]). A bimachine in the sense of [1], [7] is a bimachine as above, where λ and ρ are constant functions equal to 1.

Let $\alpha : A^* \to B^*$ be a function. We define on A^* a relation, which will be reflexive, symmetric, compatible with left multiplication, *but not transitive* in general. We call it the (*left*) *syntactic adjacency relation* of α, denoted by

$$u \leftrightarrow v.$$

It is defined by

$$\sup\{\|\alpha(fu), \alpha(fv)\|, f \in A^*, \alpha(fu) \neq \emptyset \neq \alpha(fv)\} < \infty. \tag{8}$$

Note that, in view of the definition of adjacent functions (§ 2), one has $u \leftrightarrow v$ if and only if the two functions $f \mapsto \alpha(fu)$ and $f \mapsto \alpha(fv)$ are adjacent. It is also easy to see that α is uniformly bounded if and only if $u \leftrightarrow v$ for any words u and v. Note, moreover, that if $\mathrm{dom}(\alpha) = A^*$, then $\leftrightarrow$ is transitive and equal to the left syntactic congruence of α.

We call a left congruence $\sim$ on A^* *compatible* with $\leftrightarrow$ if for any words u, v,

$$u \sim v \Rightarrow u \leftrightarrow v.$$

In terms of their graphs, this means that $\#(\sim)$ is contained in $\#(\leftrightarrow)$. Recall that when $\sim$ is a left congruence, then $R=A^*/\sim$ is naturally equipped with a left action $A^*\times R\to R$.

THEOREM 2. *Let $\alpha: A^*\to B^*$ be a rational function. Let $\sim$ be a left congruence of finite index on A^* and $R=A^*/\sim$ and r_0 the class of* 1 mod $\sim$. *The following conditions are equivalent*:

(i) *$\sim$ is compatible with the syntactic adjacency relation of α.*

(ii) *R, together with the natural left action and r_0 as initial right state, is the set of right states of some bimachine computing α.*

It will turn out that the bimachine that we obtain in the proof is completely canonical, once $\sim$ is given. Moreover, one may choose for $\sim$ the congruence considered in the previous section, thus obtaining a completely canonical bimachine. On the other hand, we shall verify that this bimachine is minimal, in the sense stated in the introduction, when α is a total function.

Proof of Theorem 2 (first part). (ii)$\Rightarrow$(i): Let R be the set of right states of a bimachine computing α. We have, by the definition of R,

$$u\sim v \Leftrightarrow ur_0=vr_0.$$

We have to show that $u\sim v$ implies (8). Suppose that $u\sim v$, that is, $ur_0=vr_0=r$, for some r in R. Let N be some integer greater than the lengths of the words (if defined) $\omega(l,u,r_0)\rho(l')$ and $\omega(l,v,r_0)\rho(l')$, for l, l' in L. We have, by (5) and (6),

$$\begin{aligned}\alpha(fu)&=\lambda(fur_0)\omega(l_0,fu,r_0)\rho(l_0fu)\\&=\lambda(fur_0)\omega(l_0,f,ur_0)\omega(l_0f,u,r_0)\rho(l_0fu)\\&=\lambda(fr)\omega(l_0,f,r)\omega(l_0f,u,r_0)\rho(l_0fu).\end{aligned}$$

Similarly,

$$\alpha(fv)=\lambda(fr)\omega(l_0,f,r)\omega(l_0f,v,r_0)\rho(l_0fv).$$

If $\alpha(fu)\neq\emptyset\neq\alpha(fv)$, then $\alpha(fu)$ and $\alpha(fv)$ have $\lambda(fr)\omega(l_0,f,r)$ as a common left factor, hence

$$\|\alpha(fu),\alpha(fv)\|<2N.$$

This shows (8), and thus $\sim$ is compatible with the left adjacency of α. $\square$

Before continuing the proof, we need several lemmas.

LEMMA 5. *If α is a rational function and $\sim$ is a left congruence on A^* of finite index, then there exist nonempty rational functions β_i, γ_i, $1\leqq i\leqq n$, such that*

(i) $\forall u, v\in A^*$, $\alpha(uv)=\bigcup_{1\leqq i\leqq n}\beta_i(u)\gamma_i(v)$.

(ii) *Each set* dom (γ_i) *is contained in a single class* mod $\sim$.

Proof. (i) follows from Lemma 1 and its proof, which show that β_i, γ_i may be chosen rational. Now, note that each class mod $\sim$ is a rational language, and that the restriction of a rational function to a rational language is still rational. So, replacing in (i) each γ_i by the union of its restrictions to each class mod $\sim$, we obtain (ii). $\square$

Remark 2. Using this lemma, it is easy to prove that *the graph of the syntactic adjacency relation of a rational function is a recognizable subset of* $A^*\times A^*$ (in the sense of [1, Chap. 3, Thm. 1.5] and [7], i.e., a finite union of sets $K\times L$, where K, L are rational languages).

Indeed, define $i\leftrightarrow j$ if the functions β_i, β_j are adjacent. Now, for $I, J\subset\{1,\cdots,n\}$, define $I\leftrightarrow J$ if for any i in I, j in J, one has $i\leftrightarrow j$. Finally, let $I(u)=\{i\mid u\in \operatorname{dom}(\gamma_i)\}$.

Then one shows that $u \leftrightarrow v$ if and only if $I(u) \leftrightarrow I(v)$. This implies that the graph of $\leftrightarrow$ is equal to

$$\bigcup_{I \leftrightarrow J} \{u \in A^* \mid I(u) = I\} \times \{v \in A^* \mid I(v) = J\},$$

which is recognizable.

We need to define the operator "longest common left factor" for sets of words rather than only pairs of words. For technical reasons, it should also be defined on the empty set. Each singleton set will be identified with its element. So, for a nonempty language L, let $\bigwedge L$ denote the longest common left factor of the words in L equal to $\emptyset$ if $L = \emptyset$.

For $x_1, \cdots, x_n \in A^* \cup \{\emptyset\}$, we define $x_1 \wedge \cdots \wedge x_n$ to be $\bigwedge L$, where L is the underlying set of the sequence. So $x_1 \wedge \cdots \wedge x_n \neq \emptyset$ if and only if at least one x_i is not equal to $\emptyset$. Note that if L is a language, then $\bigwedge L = \bigwedge L'$ for some sublanguage L' of cardinality less than or equal to 2 (indeed, if $|L| \geqq 2$, there exist words u, v in L such that $u \wedge v = \bigwedge L$). If $\alpha_1, \cdots, \alpha_n$ are functions $A^* \to B^*$, then the function $\alpha = \alpha_1 \wedge \cdots \wedge \alpha_n$ will be defined by $\alpha(f) = \alpha_1(f) \wedge \cdots \wedge \alpha_n(f)$. Note that $\operatorname{dom}(\alpha) = \bigcup_{1 \leqq i \leqq n} \operatorname{dom}(\alpha_i)$, in view of the definitions.

We shall use the easily verified identities

$$\bigwedge \left(\bigcup_{i \in I} L_i \right) = \bigwedge_{i \in I} \left(\bigwedge L_i \right)$$

for any languages L_i, $i \in I$, and

$$\bigwedge (gL) = g\left(\bigwedge L\right)$$

for any language L and g in $A^* \cup \{\emptyset\}$.

LEMMA 6. *Let $\alpha_1, \cdots, \alpha_n : A^* \to B^*$ be pairwise adjacent functions such that each α_i^{-1} preserves rationality.*

(i) *For any words g_1, g_2 in B^*, the language*

$$\{f \mid \exists w \in B^*, \alpha_1(f) = wg_1, \alpha_2(f) = wg_2\}$$

is rational.

(ii) *If the functions α_i are, moreover, rational, then $\alpha_1 \wedge \cdots \wedge \alpha_n$ is rational.*

Note that this gives an alternative proof of the following: α, α' rational and adjacent implies $\alpha \wedge \alpha'$ rational (see Proposition 1).

Remark 3. Let α_1, α_2 be as in Remark 1. Then the language $\{f \in A^* \mid \alpha_1(f) = \alpha_2(f)\}$ (this is the case $g_1 = 1 = g_2$ of the lemma) is equal to $\{f \in A^*, |f|_{a_1} = |f|_{a_2}\}$, and hence is not rational. This shows that the adjacency hypothesis is not superfluous in Lemma 6.

Proof. (i) Let p be an integer such that $|g_1|, |g_2| < p$ and that for any f in A^*, $\alpha_1(f)$ and $\alpha_2(f)$, if defined, differ only by a right factor of length less than p.

We show that for f in A^*, the condition

(a) $\quad \exists w \in B^*, \quad |w| \geqq p, \quad \alpha_1(f) = wg_1$ and $\alpha_2(f) = wg_2$

is equivalent to the condition

(b) $\quad \exists i \in \{0, \cdots, 2p-1\}, \quad \exists u \in B^p$ such that $\alpha_1(f) \in B^i(B^{2p})^* ug_1$ and $\alpha_2(f) \in B^i(B^{2p})^* ug_2$.

Suppose that this is proved. Then the language L of the lemma is equal to $L_1 \cup L_2$, where $L_1 = \{f \in A^* \mid f$ satisfies (a)$\}$ and $L_2 = \{f \in L, |\alpha_1(f)| \leqq 2p$ or $|\alpha_2(f)| \leqq 2p\}$. By the hypothesis that the α_i^{-1} preserve rationality and by (b), L_1 is rational. Moreover, if $\alpha_1(f)$ is short, then so is $\alpha_2(f)$ and vice versa. Hence, L_2 is contained in a finite union of languages of the form $L_w = \{f \in A^* \mid \alpha_1(f) = wg_1$ and $\alpha_2(f) = wg_2\}$, which are also rational; since each L_w is contained in L, we conclude that L is rational.

It is clear that (a) implies (b). Suppose that (b) holds, that is, $\alpha_1(f)=s_1ug_1$, $\alpha_2(f)=s_2ug_2$ with $|s_1|, |s_2| \equiv i \bmod 2p$. We must show that $s_1=s_2$. By adjacency, we have $\alpha_1(f)=th_1$, $\alpha_2(f)=th_2$ with $|h_1|, |h_2|<p$. As $|g_1|, |g_2|<p$, the difference between $|s_1u|$ and $|t|$ is less than p. The case is similar for $|s_2u|$ and $|t|$. Thus $\||s_1|-|s_2|\| = \||s_1u|-|s_2u|\| < 2p \Rightarrow |s_1|=|s_2|$.

Now, $|h_1| \leqq p \leqq |ug_1|$, which implies, by $s_1ug_1=th_1$, that $|s_1| \leqq |t|$, hence s_1 is a left factor of t. Similarly, s_2 is a left factor of t. As they are of equal length, they are equal.

(ii) We have, by a previous formula,

$$\alpha_1 \wedge \cdots \wedge \alpha_n = (\alpha_1 \wedge \alpha_2) \wedge \alpha_3 \wedge \cdots \wedge \alpha_n,$$

hence we may assume that $n=2$, because each α_i is adjacent to $\alpha_1 \wedge \alpha_2$.

Without loss of generality, we may assume that $\mathrm{dom}(\alpha_1)=\mathrm{dom}(\alpha_2)=D$. Then D is a finite union of languages $D(g_1, g_2)$, where $D(g_1, g_2) = \{f \in A^* \mid \exists w \in B^*, \alpha_1(f)=wg_1, \alpha_2(f)=wg_2\}$ and where g_1 and g_2 have no common left factor. Each of these languages is rational by (i), and if $f \in D(g_1, g_2)$, then $(\alpha_1 \wedge \alpha_2)(f) = \alpha_1(f)g_1^{-1}$. Hence, the restriction of $\alpha_1 \wedge \alpha_2$ to $D(g_1, g_2)$ is rational, and finally $\alpha_1 \wedge \alpha_2$ is rational, as the union of a finite number of rational functions. □

LEMMA 7. *Let $\alpha: A^* \to B^*$ be a rational function, and $\sim$ a left congruence on A^* of finite index that is compatible with the left syntactic adjacency relation of α. Let $R=A^*/\sim$ and define for each r in R a function α_r by*

$$\alpha_r(f) = \bigwedge \{\alpha(fu) \mid u \in A^*, ur_0=r\},$$

where r_0 is the class of 1 mod $\sim$. *Then there exists a finite language L_r such that*

(i) $u \in L_r \Rightarrow ur_0 = r$,

(ii) $\alpha_r(f) = \bigwedge_{u \in L_r} \alpha(fu)$.

As a consequence, the function α_r is rational.

The point of the lemma is that L_r does not depend on f (otherwise, it is immediate, using a previous remark on $\bigwedge L$).

Proof. Suppose there exists a finite language L_r such that (i) and (ii) are satisfied. By (i) and compatibility of $\sim$, the words in L_r are pairwise in relation $\leftrightarrow$, that is, the functions $f \mapsto \alpha(fu)$ are, for u in L_r, pairwise adjacent. Since these functions are rational, we obtain by (ii) and Lemma 6(ii) that α_r is a rational function.

In order to prove that there exists a finite language L_r satisfying (i) and (ii), take β_i, γ_i as in Lemma 5. By condition (ii) of this lemma, there exists for each i a unique $r(i)$ such that $u \in \mathrm{dom}(\gamma_i) \Rightarrow ur_0 = r(i)$. We know that for each i, there exists a finite language $L_i \subset \mathrm{dom}(\gamma_i)$ such that $\bigwedge \gamma_i(A^*) = \bigwedge \gamma_i(L_i)$. Let $L_r = \bigcup_{r(i)=r} L_i$. We thus have $\bigwedge \{\gamma_i(u) \mid u \in \mathrm{dom}(\gamma_i)\} = \bigwedge \{\gamma_i(u) \mid u \in L_r\}$. Moreover, (i) holds by definition. We have also

$$\begin{aligned}
\alpha_r(f) &= \bigwedge \{\alpha(fu) \mid ur_0 = r\} \\
&= \bigwedge \{\beta_i(f)\gamma_i(u) \mid ur_0 = r, 1 \leqq i \leqq n \text{ and } u \in \mathrm{dom}(\gamma_i)\} \\
&= \bigwedge \{\beta_i(f)\gamma_i(u) \mid r(i) = r, u \in \mathrm{dom}(\gamma_i)\} \\
&= \bigwedge\nolimits_{r(i)=r} (\bigwedge \{\beta_i(f)\gamma_i(u) \mid u \in \mathrm{dom}(\gamma_i)\}) \\
&= \bigwedge\nolimits_{r(i)=r} \beta_i(f)(\bigwedge \{\gamma_i(u) \mid u \in \mathrm{dom}(\gamma_i)\}) \\
&= \bigwedge\nolimits_{r(i)=r} \beta_i(f)(\bigwedge \{\gamma_i(u) \mid u \in L_r\}) \\
&= \bigwedge\nolimits_{r(i)=r} (\bigwedge \{\beta_i(f)\gamma_i(u) \mid u \in L_r\}) \\
&= \bigwedge \{\beta_i(f)\gamma_i(u) \mid r(i) = r, u \in L_r\} \\
&= \bigwedge \{\alpha(fu) \mid u \in L_r\} \quad \text{(because } u \in L_r \text{ and } u \in \mathrm{dom}(\gamma_i) \Rightarrow r(i) = r) \\
&= \bigwedge\nolimits_{u \in L_r} \alpha(fu).
\end{aligned}$$

□

LEMMA 8 (Notations of Lemma 7). *There exist a function* $\omega: A^* \times A^* \times R \to B^*$ *and a function* $\rho: A^* \to B^*$ *such that*

(i) *For any words* f, g *in* A^* *and state* r *in* R

$$\alpha_r(fg) = \alpha_{gr}(f)\omega(f, g, r);$$

(ii) *For any word* f *in* A^*

$$\alpha(f) = \alpha_{r_0}(f)\rho(f).$$

Proof. The second assertion is immediate, because by definition, $\alpha_{r_0}(f)$ is a left factor of $\alpha(f)$. If $\alpha(f) \neq \emptyset$, we define $\rho(f) = (\alpha_{r_0}(f))^{-1}\alpha(f)$. If $\alpha(f) = \emptyset$, we pose $\rho(f) = \emptyset$. Note that the set

$$\{\alpha(fgu) \mid u \in A^*, ur_0 = r\}$$

is contained in the set

$$\{\alpha(fv) \mid v \in A^*, vr_0 = gr\}.$$

Hence, by definition, $\alpha_{gr}(f)$ is a left factor of $\alpha_r(fg)$. If $\alpha_r(fg) \neq \emptyset$, we define $\omega(f, g, r) = (\alpha_{gr}(f))^{-1}\alpha_r(fg)$. If $\alpha_r(fg) = \emptyset$, we pose once again $\omega(f, g, r) = \emptyset$. □

LEMMA 9 (Notations of Lemma 7). *Define a relation* $\equiv$ *on* A^* *by*

$$f \equiv g$$

if and only if

$$\omega(fu, a, r) = \omega(gu, a, r)$$

for any word $u \in A^*$, *letter* $a \in A$, *and state* r *in* R, *and if*

$$\rho(fu) = \rho(gu)$$

for any word u. *Then* $\equiv$ *is a right congruence of finite index.*

Proof. Recall that when $\delta_1, \cdots, \delta_p$ are functions $A^* \to B^*$ such that

(i) Each $\delta_i(A^*)$ is finite;

(ii) For each g in B^* and i, $\delta_i^{-1}(g)$ is a rational language;

then by Nerode's criterion, the right congruence on A^*, defined by $f \equiv g$ if and only if $\delta_i(fu) = \delta_i(gu)$ for any i and u, is of finite index.

Hence, it is enough to show that the functions $\omega(\cdot, a, r): f \mapsto \omega(f, a, r)$ and ρ have finite image and that for any a, r, g in B^*, the languages $\{f \in A^* \mid \omega(f, a, r) = g\}$ and $\{f \in A^* \mid \rho(f) = g\}$ are rational.

For this, it is enough, in view of Lemma 6(i) and Lemma 7, to show that the functions $f \mapsto \alpha_r(fa)$ and $f \mapsto \alpha_{ar}(f)$ are adjacent for any $a \in A$ and $r \in R$, and that the functions α and α_{r_0} are adjacent.

By Lemma 7, we have $\alpha_r(fa) = \bigwedge_{u \in L_r} \alpha(fau)$ and $\alpha_{ar}(f) = \bigwedge_{v \in L_{ar}} \alpha(fv)$.

Note that $u \in L_r$ and $v \in L_{ar}$ implies that $ur_0 = r \Rightarrow aur_0 = ar$, and $vr_0 = ar$. Hence $au \sim v$, which implies $au \leftrightarrow v$ and the functions $f \mapsto \alpha(fau)$ and $f \mapsto \alpha(fv)$ are adjacent. Moreover, for $w, w' \in L_r$, one has $w \sim w'$ (by Lemma 7 (i)), hence $w \leftrightarrow w'$ (by compatibility of $\sim$), hence the functions $f \mapsto \alpha(fw)$ and $f \mapsto \alpha(fw')$ are adjacent. This shows that the functions $f \mapsto \alpha_r(fa)$ and $f \mapsto \alpha_{ar}(f)$ are adjacent, because of the following easily verified fact: if $\alpha_1, \cdots, \alpha_n$ (respectively, $\beta_1, \cdots, \beta_p$) are pairwise adjacent, and if each α_i is adjacent to each β_j, then $\alpha_1 \wedge \cdots \wedge \alpha_n$ is adjacent to $\beta_1 \wedge \cdots \wedge \beta_p$.

Moreover, $\alpha_{r_0}(f) = \bigwedge_{u \in L_{r_0}} \alpha(fu)$ and a similar proof shows that this function is adjacent to α. □

Proof of Theorem 2 (Second part). Let $L = A^*/\equiv$, where $\equiv$ is the right congruence of Lemma 9. Then L is finite, and equipped with a right action $L \times A^* \to L$. For l in L, a in A, and r in R, we may define $\omega(l, a, r) = \omega(f, a, r)$ and $\rho(l) = \rho(f)$, where f is a representative of $l \bmod \equiv$.

Let l_0 be the class of $1 \bmod \equiv$. Define a function $\lambda: R \to B^*$ by $\lambda(r) = \alpha_r(1)$.

With these pointed sets (L, l_0), (R, r_0) and functions ω, λ, ρ, we obtain a bimachine for which we have only to verify that it computes α, that is, formulas (5) and (6). For this, it is enough to show that the functions ω and ρ of Lemma 8 satisfy

$$\omega(f, gh, r) = \omega(f, g, hr)\omega(fg, h, r) \tag{9}$$

and

$$\alpha(f) = \lambda(fr_0)\omega(1, f, r_0)\rho(f). \tag{10}$$

But we have, by Lemma 8,

$$\alpha_r(fgh) = \alpha_{ghr}(f)\omega(f, gh, r)$$

and

$$\begin{aligned}\alpha_r(fgh) &= \alpha_{hr}(fg)\omega(fg, h, r)\\ &= \alpha_{ghr}(f)\omega(f, g, hr)\omega(fg, h, r).\end{aligned}$$

So (9) is true as soon as $\alpha_{ghr}(f) \neq \emptyset$. When $\alpha_{ghr}(f) = \emptyset$, then $\alpha_r(fgh) = \emptyset$, and by the definition of ω, we have $\omega(fg, h, r) = \emptyset = \omega(f, gh, r)$. So, (9) is also true.

For (10), we have, by Lemma 8,

$$\alpha(f) = \alpha_{r_0}(f)\rho(f) = \alpha_{r_0}(1 \cdot f)\rho(f) = \alpha_{fr_0}(1)\omega(1, f, r_0)\rho(f) = \lambda(fr_0)\omega(1, f, r_0)\rho(f),$$

which proves (10). □

Remark 4. (1) Note that when r_0 in R is replaced by r, and ρ by the constant function ρ' equal to 1, then this new bimachine computes α_r. Indeed, by Lemma 8,

$$\alpha_r(f) = \alpha_r(1 \cdot f) = \alpha_{fr}(1)\omega(1, f, r) = \lambda(fr)\omega(l_0, f, r)\rho'(l_0 f).$$

(2) When α is a subsequential function, then its left syntactic adjacency is universal (i.e., $u \leftrightarrow v$ for any word u, v), hence a left congruence. If one takes this congruence for $\sim$ in Theorem 2, then the bimachine constructed in the proof is exactly the minimal subsequential transducer of α, as constructed by Choffrut [4] (see also [11]).

5. Example, remarks, and open problems. (a) Let $A = \{a, b\}$ and $\alpha: A^* \to A^*$ be the function which removes odd runs in a word. More formally, if

$$w = a^{i_1}b^{j_1} \cdots a^{i_k}b^{j_k}$$

where the exponents are greater than or equal to 1, except possibly i_1 and j_k, then define

$$i'_s = \begin{cases} i_s & \text{if } i_s \text{ is even}\\ 0 & \text{otherwise;}\end{cases}$$

$$j'_s = \begin{cases} j_s & \text{if } j_s \text{ is even}\\ 0 & \text{otherwise.}\end{cases}$$

Then $\alpha(w) = a^{i'_1}b^{j'_1} \cdots a^{i'_k}b^{j'_k}$. Moreover, $\alpha(1) = 1$.

We leave to the reader the verification of the following facts.

(1) The left syntactic congruence $\sim$ of α is generated by the relations

$$a^2 \sim 1, \qquad b^2 \sim 1, \qquad ab \sim a, \qquad ba \sim b.$$

(2) Identify $R = A^*/\sim$ with $\{1, a, b\}$. The functions $\alpha_1, \alpha_a, \alpha_b$ are defined by $\alpha_1 = \alpha$, $\alpha_a(f) = \alpha(fa)$, $\alpha_b(f) = \alpha(fb)$.

(3) The function ρ is constant and equal to 1, and

$$\omega(f, a, 1) = \omega(f, b, 1) = \omega(f, a, b) = \omega(f, b, a) = 1.$$

Moreover,

$$\omega(f, a, a) = \begin{cases} a^2 & \text{if the last run of } f \text{ is an even run of } a\text{'s or if} \\ & f \text{ does not end with } a. \\ 1 & \text{otherwise;} \end{cases}$$

$$\omega(f, b, b) = \begin{cases} b^2 & \text{if the last run of } f \text{ is an even run of } b\text{'s or if} \\ & f \text{ does not end with } b. \\ 1 & \text{otherwise.} \end{cases}$$

(4) The right congruence $\equiv$ is generated by the relations

$$a^2 \equiv 1, \qquad b^2 \equiv 1, \qquad ab \equiv b, \qquad ba \equiv a.$$

Actually, it is the *right* syntactic congruence of α (this is not a general fact, even for everywhere-defined functions).

(5) The function λ is constant equal to 1 and if $L = A^*/\equiv$ is identified with $\{1, a, b\}$, then ω is described by the following tables

$l \backslash r$	1	a	b
1	1	a^2	1
a	1	1	1
b	1	a^2	1

$\omega(l, a, r)$

$l \backslash r$	1	a	b
1	1	1	b^2
a	1	1	b^2
b	1	1	1

$\omega(l, b, r)$

5.1. Minimization. We verify that, when α is a *total function*, then the bimachine constructed in the proof of Theorem 2 has the minimum number of left states among all bimachines computing α, with R as a set of right states (with its natural left action), with r_0 as initial right state.

So let α be computed by the bimachine B' with a set of left states L', initial left state l'_0, set of right states R, initial right state r_0, output function ω', final left function λ', and final right function ρ'.

We show that for any words g, f in A^*, the equality $l'_0 f = l'_0 g$ implies $f \equiv g$ (where $\equiv$ is the right congruence of Lemma 9). This will imply that $L = A^*/\equiv$ has fewer elements than $l'_0 A^*$, hence fewer than L' (because $l'_0 A^* \subset L$).

We work in the free group generated by A. With the notations of Lemma 7, we have

$$\alpha_r(f) = \bigwedge \{\alpha(fu) \mid u \in A^*, ur_0 = r\}. \tag{11}$$

By (5) and (6) applied to bimachine B', we have

$$\alpha(fu) = \lambda'(fur_0)\omega'(l'_0, f, ur_0)\omega'(l'_0 f, u, r_0)\rho'(l'_0 fu).$$

This, along with (11), implies that

$$\alpha_r(f) = \lambda'(fr)\omega'(l'_0, f, r)\beta(l'_0 f, r) \tag{12}$$

where β: $L' \times R \to B^*$ is the function defined by

$$\beta(l', r) = \bigwedge \{\omega'(l', u, r_0)\rho'(l'u) \mid ur_0 = r\}.$$

From (12) we deduce

$$\begin{aligned}\alpha_r(fg) &= \lambda'(fgr)\omega'(l'_0, fg, r)\beta(l'_0fg, r) \\ &= \lambda'(fgr)\omega'(l'_0, f, gr)\omega'(l'_0f, g, r)\beta(l'_0fg, r),\end{aligned} \tag{13}$$

where we have used (5) again. From (12) again, we deduce

$$\alpha_{gr}(f)\omega(f, g, r) = \lambda'(fgr)\omega'(l'_0, f, gr)\beta(l'_0f, gr)\omega(f, g, r). \tag{14}$$

Recall that we have, by Lemma 8(i),

$$\alpha_r(fg) = \alpha_{gr}(f)\omega(f, g, r).$$

Using this and comparing (13) and (14), we therefore deduce that

$$\omega(f, g, r) = \beta(l'_0f, gr)^{-1}\omega'(l'_0f, g, r)\beta(l'_0fg, r). \tag{15}$$

Indeed, α is a total function, so α_r and α_{gr} are total functions as well, and every factor in (13) and (14) is defined; we thus may simplify by $\lambda'(fgr)\omega'(l'_0, f, gr)$, and multiply (in the free group) by $\beta(l'_0f, gr)^{-1}$.

By Lemma 8(ii), we have

$$\alpha(f) = \alpha_{r_0}(f)\rho(f).$$

As α is computed by β', and by (12), we thus obtain

$$\lambda'(fr_0)\omega'(l'_0, f, r_0)\rho'(l_0f) = \lambda'(fr_0)\omega'(l'_0, f, r_0)\beta(l'_0f, r_0)\rho(f).$$

Thus, we deduce

$$\rho(f) = \beta(l'_0f, r_0)^{-1}\rho'(l_0f). \tag{16}$$

Now, let f, g, u, a, r be as in Lemma 9, and suppose that $l'_0f = l'_0g$. Then by (15), used twice (with $f \to fu$, $g \to a$, and after $f \to gu$, $g \to a$), we obtain

$$\begin{aligned}\omega(fu, a, r) &= \beta(l'_0fu, ar)^{-1}\omega'(l'_0fu, a, r)\beta(l'_0fua, r) \\ &= \beta(l'_0gu, ar)^{-1}\omega'(l'_0gu, a, r)\beta(l'_0gua, r) = \omega(gu, a, r).\end{aligned}$$

Moreover, by (16), we have

$$\begin{aligned}\rho(fu) &= \beta(l'_0fu, r_0)^{-1}\rho'(l_0fu) \\ &= \beta(l'_0gu, r_0)^{-1}\rho'(l_0gu) = \rho(gu).\end{aligned}$$

This shows, by Lemma 9, that $f \equiv g$, which was to be shown.

5.2. Counterexample. We show that when α is not a total function, then the minimization result of § 5.1 is no longer valid. This is a mystery which should be elucidated elsewhere.

Let $\alpha : a^* \to a^*$ be defined by $\alpha(a^{2n}) = a^{2n}$, $\alpha(a^{2n+1}) = \emptyset$. Take $R = a^*/a^2 \sim 1$ ($\sim$ is the syntactic left congruence) and identify R with $\{1, a\}$. Then

$$\begin{aligned}
\alpha_1(1) &= \bigwedge \{\alpha(u), u.1 = 1\} \\
&= \bigwedge \{\alpha(a^{2n}), n \in \mathbb{N}\} = 1 \\
\alpha_1(a) &= \bigwedge \{\alpha(au), u.1 = 1\} \\
&= \bigwedge \{\alpha(aa^{2n}), n \in \mathbb{N}\} = \emptyset \\
\alpha_a(a^2) &= \bigwedge \{\alpha(a^2u), u.1 = a\} \\
&= \bigwedge \{\alpha(a^2a^{2n+1}), n \in \mathbb{N}\} = \emptyset \\
\alpha_a(a) &= \bigwedge \{\alpha(au), u.1 = a\} \\
&= \bigwedge \{\alpha(aa^{2n+1}), n \in \mathbb{N}\} = a^2.
\end{aligned}$$

Using Lemma 8, we have $\alpha_a(a) = \alpha_1(1)\ \omega(1, a, a)$ and $\alpha_a(a^2) = \alpha_1(a)\ \omega(a, a, a)$. Hence, $\omega(1, a, a) = a^2$, and $\omega(a, a, a) = \emptyset$ (see the proof of Lemma 8). We deduce, by Lemma 9, that $a \not\equiv 1$.

However, the function is subsequential in both directions, hence, it may be computed with R as a set of right states, and a trivial set of left states (i.e., a singleton).

The reader may find it instructive to compare the previous example to the two following ones:

$$\begin{cases} a^{2n} \to a^{2n} \\ a^{2n+1} \to b^{2n+1} \end{cases} \qquad \begin{cases} a^{2n} \to 1 \\ a^{2n+1} \to a \end{cases}.$$

The first function is not subsequential, in either direction, while the second is subsequential in both directions.

5.3. Open problem. A theory of morphisms between bimachines computing the same function α should be developed, keeping in mind the following possible conjecture: there are only a finite number of minimal bimachines computing α (minimal would mean universally attractive in the category of these bimachines).

One cannot expect a single minimal bimachine: evidence for this is given by the rational languages; there is no "morphic" relation between the left and the right minimal automaton.

5.4. Open problem. A bimachine has two sets of states, hence there are two finite monoids attached to it. Call a bimachine *aperiodic* such that these monoids are aperiodic (i.e., with trivial subgroups, or period equal to 1). Characterize the rational functions α, which are computed by some aperiodic bimachine. A tentative conjecture could be: α is as above if and only if for any rational language L, the period of $\alpha^{-1}(L)$ divides that of L (recall that p is a period of L if the cardinality of each cyclic subgroup of the syntactic monoid of L divides p).

More generally, a theory of varieties of rational functions could be made, as has been done for rational languages and finite monoids [10]. A first step would be to study sequential and subsequential functions.

5.5. Open problem. Characterize rational functions which are both left-to-right and right-to-left subsequential. These functions simultaneously generalize rational languages (by their characteristic function) and biprefix codes (by their decoding functions).

An answer in the case of numerical functions (i.e., with image in a cyclic free monoid) has been given by Choffrut and Schützenberger [6].

Acknowledgments. We want to thank the two referees for many valuable comments and suggestions.

REFERENCES

[1] J. BERSTEL, *Transductions and Context-Free Languages*, Teubner, Stuttgart, Germany, 1979.
[2] J. BERSTEL AND D. PERRIN, *Theory of Codes*, Academic Press, New York, 1985.
[3] J.-M. BOË, J. BOYAT, J.-P. BORDAT, AND Y. CÉSARI, *Une caractérisation des sous-monoïdes libérables*, in Théorie des codes, D. Perrin, ed., Laboratoire d'Informatique Théorique et de Programmation, Paris, (1979), pp. 9-20.
[4] C. CHOFFRUT, *A generalization of Ginsburg and Rose's characterization of g.-s.-m. mappings*, Lecture Notes in Computer Science 71, Springer-Verlag, Berlin, New York, 1979, pp. 88-103.
[5] C. CHOFFRUT AND K. CULIK, *Properties of finite and push-down transducers*, SIAM J. Comput., 12 (1983), pp. 300-315.
[6] C. CHOFFRUT AND M. P. SCHUTZENBERGER, *Counting with rational functions*, Theoret. Comput. Sci., 58 (1988), pp. 81-101.
[7] S. EILENBERG, *Automata, Languages and Machines*, Vol. A, Academic Press, New York, 1974.
[8] S. GINSBURG, *An Introduction to Mathematical Machine Theory*, Addison-Wesley, Reading, MA, 1962.
[9] M. LOTHAIRE, *Combinatorics on Words*, Addison-Wesley, Reading, MA, 1983.
[10] J.-E. PIN, *Variétés de langages formels*, Masson, Paris, 1984.
[11] C. REUTENAUER, *Subsequential functions: Characterizations, minimization, examples*, in Proc. International Meeting of Young Computer Scientists, Lecture Notes in Computer Science, J. Kelemen, ed., to appear.
[12] M. P. SCHUTZENBERGER, *A remark on finite transducers*, Inform. and Control, 4 (1961), pp. 185-196.
[13] ———, *Une propriété de Hankel des relations fonctionnelles entre monoïdes libres*, Adv. in Math., 24 (1977), pp. 274-280.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 686-694, August 1991

© 1991 Society for Industrial and Applied Mathematics
006

THE FAST m-TRANSFORM: A FAST COMPUTATION OF CROSS-CORRELATIONS WITH BINARY m-SEQUENCES*

ERICH E. SUTTER†

Abstract. An algorithm is presented for the fast computation of the m-transform, a Hadamard transform intimately related to cross-correlation of analog signals with binary m-sequences. It is shown that m-transforms are in the same Hadamard equivalence class as Walsh–Hadamard transforms and can, thus, be computed by means of the Fast Walsh Transform (FWT) algorithm, preceded and followed by a permutation. The FWT is performed in place in the original data array, while the permutations are executed during loading and reading of this array. Real-time generation of the array addresses for loading and reading adds little to execution time of the FWT. The implementation described here lends itself particularly well to applications in linear and nonlinear systems analysis.

Key words. fast cross-correlation, Hadamard transforms, m-sequences, nonlinear systems analysis, Walsh–Hadamard transforms

AMS(MOS) subject classifications. 65F30, 65C25, 05B20, 42C10

Introduction. The first theoretical work on binary m-sequences was published by Zierler in 1959 [1]. During the following decades their properties were extensively studied [2]. Researchers soon found applications in the fields of systems analysis and identification. The ease and speed with which these pseudorandom sequences could be generated made them very attractive in situations where random white processes are called for. In one of the first such applications, Briggs et al. [3] used them for a linear correlation analysis of process dynamics. Subsequently, numerous applications in nonlinear systems analysis were explored [4], [5], [6]. It was discovered, however, that the randomness properties of m-sequences, as exhibited in higher order auto-correlation functions, are not adequate for emulation of truly random sequences [6], [7], [8]. Detailed studies of their auto-correlation properties [9] ultimately discredited binary m-sequences as test inputs for stochastic white noise of nonlinear systems. The recent introduction of a deterministic technique [10], however, renewed interest in the application of binary m-sequences to systems analysis problems. In this new approach, the derivation of the binary kernels of all orders is reduced to a single cross-correlation of the binary m-sequence test input and the corresponding output. It is necessary, however, that the test extend over a long, complete m-sequence cycle, and that the entire cross-correlation cycle be computed. Because of the often very large size of the arrays, selection of the right algorithm can be very important. Traditionally, such cases called for application of the convolution theorem, requiring execution of three Fast Fourier Transforms (FFTs). In this case, however, where one of the arrays is a binary sequence of a specific class, a much faster computational technique is possible. As shown below, the computation can be reduced to a single Fast Walsh Transform (FWT).

1. Background.

1.1. Hadamard bases and Walsh–Hadamard transforms.

A Hadamard matrix is an orthogonal $m \times m$ matrix whose elements are binary $(+1, -1)$. The linear transform mediated by the Hadamard matrix is called a Hadamard transform. Orthogonality requires that the dimensionality be even. The rows or columns of the matrix are orthogonal binary vectors in an m-dimensional vector space.

$$\mathbf{H}\mathbf{H}^T = \mathbf{H}^T\mathbf{H} = m \cdot \mathbf{I}. \tag{1}$$

* Received by the editors August 21, 1989; accepted for publication (in revised form) October 30, 1990.

† Smith-Kettlewell Eye Research Institute, 2232 Webster St., San Francisco, California 94115.

Clearly, multiplication with -1 and permutations of rows and columns cannot affect this property. Any two Hadamard matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ are said to be equivalent if

$$\mathbf{H}_2 = \mathbf{P}_r^T \mathbf{H}_1 \mathbf{P}_c, \tag{2}$$

where $\mathbf{P}_c$ and $\mathbf{P}_r$ are permutation matrices for columns and rows, respectively.

The existence of different equivalence classes has been demonstrated by Hall [11] for the special cases $m = 16$ and $m = 20$.

Of special interest here are Hadamard matrices of order 2^n. For each $m = 2^n$ there exists at least one equivalence class that contains the different representations of the Walsh transform matrix. The Walsh matrices can be defined in various ways leading to different orderings of the Walsh vectors (see, e.g., [12]). The representation considered here is called natural, or Hadamard, ordering. It is achieved by means of a pair of binary registers of length n. These registers, C and R, contain the binary representation of the row number r and column number c, respectively. Let r_i and c_i be the digits of the binary registers C and R, respectively. The Walsh matrix is then given by

$$W(c, r) = (-1)^{q(c,r)} \quad \text{where} \quad q(c, r) = \sum_{i=0}^{n-1} r_i c_i. \tag{3}$$

Each matrix element is thus defined as the parity of the bitwise logic AND between a register r, containing its row number, and a register c, containing the column number.

Example. $n = 3$.

Matrix element $W_{5,6}$:

$$\left.\begin{array}{ll} \textit{row} & r = 5\colon \{r_i\} \to (101) \\ \textit{column} & c = 6\colon \{c_i\} \to (110) \end{array}\right\} \text{AND} \to (001) \to \textit{parity} \to W(5, 6) = -1. \tag{4}$$

The entire matrix is

(5)

r \ c	0	1	2	3	4	5	6	7		
0	+	+	+	+	+	+	+	+	=	$\vec{w}_0$
1	+	−	+	−	+	−	+	−	=	$\vec{w}_1$
2	+	+	−	−	+	+	−	−	=	$\vec{w}_2$
3	+	−	−	+	+	−	−	+	=	$\vec{w}_3$
4	+	+	+	+	−	−	−	−	=	$\vec{w}_4$
5	+	−	+	−	−	+	−	+	=	$\vec{w}_5$
6	+	+	−	−	−	−	+	+	=	$\vec{w}_6$
7	+	−	−	+	−	+	+	−	=	$\vec{w}_7$

All other Walsh–Hadamard matrix representations of the same order are obtained by permutation of the row and column numbers, and are, thus, in the same equivalence class according to equivalence relation (2):

$$W'(c, r) = W(p_2(c), p_1(r)), \tag{6}$$

where p_1 and p_2 are permutation operators.

The Walsh–Hadamard transform, in its natural ordering as defined by (3), can be computed by a simple Fast Walsh Transform (FWT) algorithm [13]. Similar fast algorithms have been developed for various other orderings [12]. According to (2),

any other transform of this equivalence class can be computed by means of the same algorithm preceded and followed by a permutation matrix. Techniques of transformation from one ordering to another have also been developed [14].

1.2. Binary m-sequences. Binary m-sequences, also called maximal length shift register sequences, can be generated by means of digital shift registers with feedback, as shown in Fig. 1 [2]. The content of a certain set of register stages is summed modulo 2 and fed back into the input.

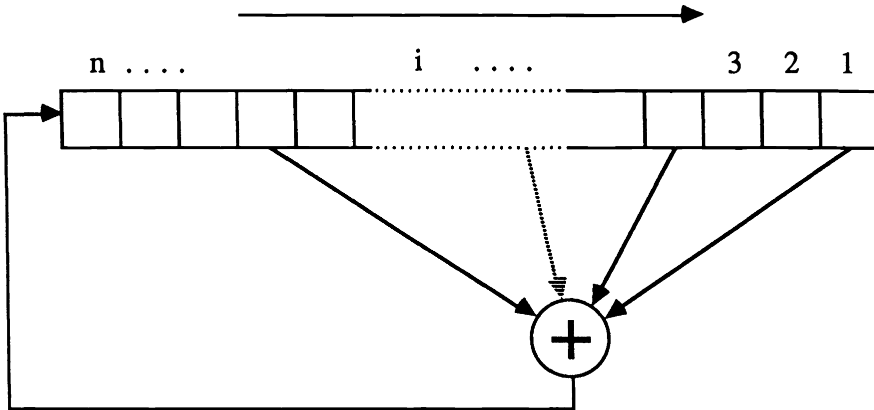


FIG. 1

With properly chosen feedback taps, the register cycles through all possible configurations, except for the all-zero configuration, which is a cycle in itself. For the larger of the two cycles, the binary sequence of 0's and 1's generated by the output of the register is called a maximal length shift register sequence or binary m-sequence. It follows immediately that:

(1) m-sequences have a period of 2^n-1, where n is the number of stages in the generating register.

(2) The number of 1's exceeds the number of 0's by exactly one, i.e.,

$$\sum_{i=0}^{2^n-1} a_i = 2^{n-1}. \tag{7}$$

These sequences have been extensively studied [1], [2], [6], [9].

Let $A_1=\{a_1, a_2, a_3, \cdots\}$ be a binary m-sequence with period 2^n-1 and $A_i=\{a_i, a_{i+1}, a_{i+2}, \cdots\}$ be the sequence in all its cyclical shifts. Let $A_0=\{0, 0, 0, \cdots\}$.

$$\begin{array}{lclllllll} A_0 & = & 0 & 0 & 0 & \cdot & \cdot & 0 & \\ A_1 & = & a_1 & a_2 & a_3 & \cdot & \cdot & a_N & \\ A_2 & = & a_2 & a_3 & a_4 & \cdot & \cdot & a_1 & \\ A_3 & = & a_3 & a_4 & a_5 & \cdot & \cdot & a_2 & \text{where } N=2^n-1. \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \\ A_N & = & a_{N-1} & \cdot & \cdot & \cdot & \cdot & a_{N-1} & \end{array} \tag{8}$$

The sequences $A_0, A_1, A_2, \cdots, A_p$ form an Abelian group with respect to the operation of elementwise addition modulo 2. Specifically

$$A_i+A_0=A_i, \qquad A_i+A_i=A_0 \quad \text{for any } i \quad \text{and} \quad A_i+A_j=A_{k(i,j)} \quad \text{for } i\neq j\neq 0. \tag{9}$$

The proof follows directly from the recurrence relation defined by Fig. 1 (see [2, p. 44]).

2. Binary m-transform.

DEFINITION. Let $\{M_i\}$ be the set of sequences obtained from the sequences $\{A_i\}$ by replacing all the 0's by 1's and the 1's by -1's, and adding a zeroth element of 1 to each A_i.

$$\text{(10)} \qquad \mathbf{M} = \begin{bmatrix} M_0 \\ M_1 \\ M_2 \\ M_3 \\ \cdot \\ \cdot \\ M_N \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & m_1 & m_2 & \cdot & \cdot & m_N \\ 1 & m_2 & m_3 & \cdot & \cdot & m_1 \\ 1 & m_3 & m_4 & \cdot & \cdot & m_2 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & m_N & m_1 & \cdot & \cdot & m_{N-1} \end{bmatrix} \qquad N = 2^n - 1.$$

The transform defined by matrix (10) will be called m-transform.

Through the substitution $0 \to 1, 1 \to -1$, the operation of addition modulo 2 becomes multiplication:

$$\text{(11)} \qquad \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \qquad \begin{array}{l} 0 \to 1 \\ 1 \to -1 \end{array} \qquad \begin{array}{c|cc} & 1 & -1 \\ \hline 1 & 1 & -1 \\ -1 & -1 & 1 \end{array}.$$

With the completion of the rows with a zeroth element of 1, the zero cycle is included in each row. With it, the matrix becomes symmetrical. From (7) and (9) it follows that the rows M_r form an orthogonal basis

$$\text{(12)} \qquad \vec{M}_r \cdot \vec{M}_s = \begin{cases} 2^n & \text{for } i = k \\ 0 & \text{for } i \neq k, \end{cases}$$

and that $\mathbf{M}$ is a symmetric orthogonal matrix

$$\text{(13)} \qquad \mathbf{M}^T\mathbf{M} = \mathbf{M}\mathbf{M} = 2^n \cdot \mathbf{I} \quad \text{where } \mathbf{I} \text{ is the identity.}$$

With the above substitution, the rows M_r now form an Abelian group with respect to elementwise multiplication.

As a binary orthogonal matrix, $\mathbf{M}$ is a Hadamard matrix.

Note that the cross-correlation of a data array of $2^n - 1$ real numbers with a binary m-sequence (elements $+1$ and -1) is the sequence of elements 1 to $2^n - 1$ of the m-transform if the data array is supplemented with a zeroth element of 0.

THEOREM. *All Walsh and* m-*transform matrices of dimension* 2^n *are in the same equivalence class of Hadamard matrices.*

Proof. Each row M_r of the matrix $\mathbf{M}$ can be obtained as the parity of a particular collection of taps t_r on its n stages during a single cycle through the configurations of the generating register. This can be seen as follows. For the first n rows, t_r is just a single tap on the rth stage. For row M_{n+1}, a single tap on stage $n+1$ would be needed. According to Fig. 1, this tap is equivalent to the configuration t_{n+1} of the feedback taps (see Fig. 2).

For $M_{n+2}, M_{n+3}, \cdots$, the feedback taps have to be shifted left one stage each time. Whenever a tap is shifted off the left end of the register, it is replaced by the feedback tap configuration. If, in this process, a new tap coincides with an already existing tap, this tap position contributes even parity and can, thus, be dropped.

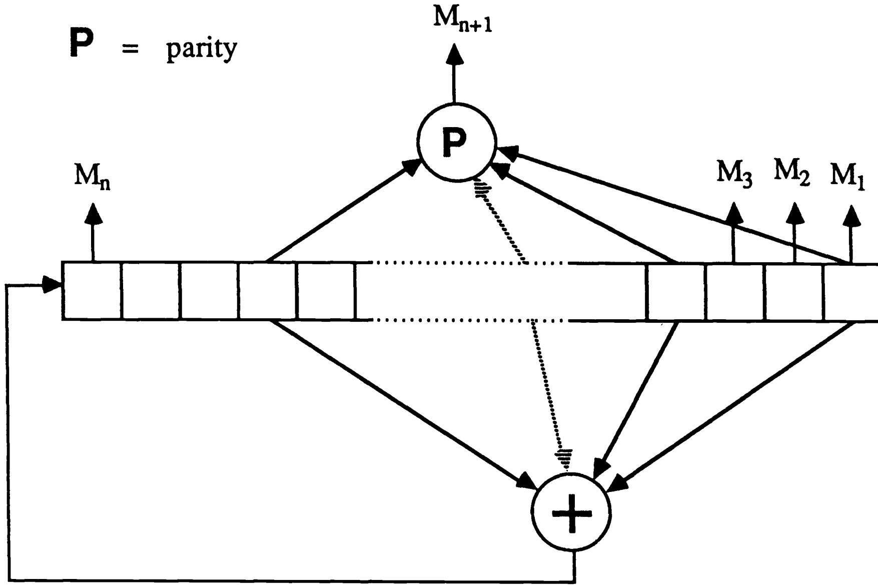


FIG. 2

The generation of the tap configurations t_r derived above can be implemented in a separate shift register of the same length n. It will be called tap register T. The 1's in this register signify the position of taps. The tap register is initialized with a right-justified 1 for the first row and shifted left for consecutive rows. The output of the register is added (modulo 2) to the stages where the generating register has its feedback taps (see top of Fig. 3).

The tap configurations sequentially generated by this method will produce consecutive rows of the matrix **M**. Similarly, for a fixed configuration of the generating register, the set of all tap configurations yields a column. With each shift of the generating register, a new column is generated.

So, to get to the matrix element $M_{c,r}$, the tap and generating registers are advanced by $(c-1)$ and $(r-1)$ steps, respectively. $M_{c,r}$ is then the parity of the bitwise logic AND (tap operation) of the two registers. For the generation of the zeroth row (zeroth column), the column (row) register is initialized with all 0's.

Note that a row generated by a collection t_r of taps is simply the bitwise product of all the rows generated by the individual taps in the collection. It follows that the Abelian group of rows is generated by rows M_1 through M_n in the same way as the Walsh basis is generated by the Rademacher functions [15].

This derivation of the m-transform matrix serves as another definition of the m-transform in terms of the generating and tap registers.

$$M(r, c) = (-1)^{q(r,c)} \quad \text{where } q(r, c) = \sum_{i=0}^{n-1} t_i(r) g_i(c), \tag{14}$$

where $g_i(c)$ and $t_i(r)$ are cth and rth bit configurations generated by the registers G and T, respectively, and $g_i(0) = t_i(0) = 0$. For the generation of the nontrivial cycle, the registers G and T can be initialized with any binary number not equal to 0, depending on the chosen starting point of the m-sequence. In applications to deterministic nonlinear analysis, the register T takes on a special function that determines the initialization [10].

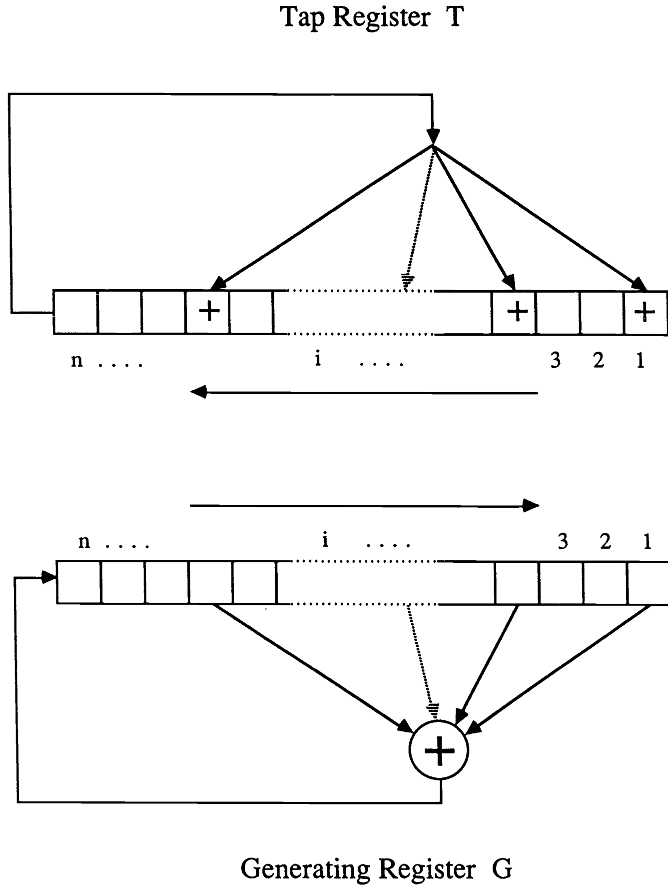


FIG. 3

This derivation of the matrix **M** matches the definition of the natural Walsh transform matrix **W** (3), except in the sequence in which the $2^n - 1$ register configurations are being generated. It, thus, follows that

$$M(r, c) = W(t(r), g(c)), \tag{15}$$

i.e., m-transforms and Walsh transforms belong to the same equivalence class of Hadamard transforms.

2.1. Fast computation of m-transforms. The equivalence between Walsh and m-transforms makes it possible to compute m-transforms by means of the Fast Walsh Transform (FWT) algorithm using natural (Hadamard) ordering. The permutations $r \to g(r)$ and $c \to t(c)$ preceding and following the FWT do not add significantly to the computation times. This section discusses efficient execution of these permutations.

In matrix notation, (15) can be written as

$$\mathbf{M} = \mathbf{P}_{r\to t}^T \mathbf{W} \mathbf{P}_{g\to c}, \tag{16}$$

where $\mathbf{P}_{c\to g}$ is the permutation matrix $c \to g(c)$ defined by the generating register, and $\mathbf{P}_{r\to t}$ is the permutation matrix $r \to t(r)$ defined by the tap register.

From the symmetry of the matrices **M** and **W**, it follows that

$$\mathbf{M}^T = \mathbf{P}_{c\to g}^T \mathbf{W}^T \mathbf{P}_{r\to t} = \mathbf{P}_{c\to g}^T \mathbf{W} \mathbf{P}_{r\to t} = \mathbf{M}, \tag{17}$$

i.e., the roles of the tap and generating registers can be interchanged. The sequence of operations chosen here has important advantages in applications to nonlinear systems analysis [10].

Fig. 4 illustrates the relationship between the registers and matrices for the case $n=3$.

The permutation $\mathbf{P}_{c\to g}$ is equivalent to loading data point number c at the cth binary array address generated by register G. The permutation $\mathbf{P}^T_{r\to t}$, after execution of FWT, is equivalent to reading point number r of the m-transform from the rth binary array address generated by the tap register T.

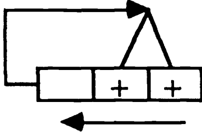
It is, of course, possible to compute the two address arrays for a particular m-transform ahead of time. However, the generation of the addresses is considerably faster than loading from a conventional storage medium, particularly if the instruction set of processors contains the register operation of bitwise exclusive OR (EXOR). Consecutive configurations of the register T (addresses for retrieval of the m-transform) can be generated at high speed using the following simple operations. (1) Shift register T left by one. (2) If bit $n+1$ of register T is set, then $T \equiv T$ EXOR C, where the register C contains 1's bit position $(n+1)$, as well as the position of feedback taps and 0's everywhere else.

Since each bit of register T cycles through the m-sequence, the same code can be used to generate consecutive addresses for loading of the data points. The output of T is simply shifted from the left through the n least significant bit positions of another register G.

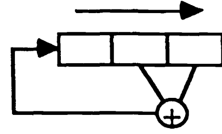
3. Discussion and conclusions. Among the Hadamard sets, the m-sequence bases are unique in that they share the following two important properties. First, all m-sequence basis vectors are related to one another by cyclical shifts of the elements 1 through 2^n-1. Second, the basis vectors form an Abelian group with respect to elementwise multiplication. These properties make them extremely valuable as test inputs for the analysis of nonlinear systems. They make it possible to reduce the data analysis to a single cross-correlation between the system response and the m-sequence input [10]. Since these two arrays can be very large, efficient computation of the cross-correlation cycle is of great importance. The technique presented here reduces the computation to a single Fast Walsh Transform that is performed in-place. The reduction in computation time, compared to the traditional method employing FFTs and the convolution theorem, is considerable. Three FFTs and an array multiplication are replaced by a single Fast Walsh Transform (FWT) preceded and followed by simple and highly efficient routines for loading and unloading of the data array. The loading and unloading routines require little or no overhead, depending on the application. The FWT algorithm is basically an abbreviated FFT, requiring no sine table and no multiplications. An exact quantitative measure of the speed advantage of the FWT over the FFT cannot be given, since it depends on the available hardware. In most cases, it can easily be implemented in integer, rather than floating point format without loss of accuracy. In a test on a Macintosh II computer using the 68081 math co-processor, the FWT was faster than a real FFT by a factor of six. Both transforms used in the comparison were based on the Cooley–Tukey algorithm. Since the computation of the cross-correlation cycle requires only one FWT, one can expect an overall speed advantage of a factor between 15 and 30. On systems without hardware multiplier, the savings are significantly larger.

Consider also that the Fast m-transform makes use of the fact that the m-sequence is completely determined by the length of the generating register and the configuration

Tap Register T



Generating Register G



r		$t_1(r)$		t
0	→	000	→	0
1	→	001	→	1
2	→	010	→	2
3	→	100	→	4
4	→	011	→	3
5	→	110	→	6
6	→	111	→	7
7	→	101	→	5

zero cycle (row 0)

m-sequence cycle (rows 1–7)

c		$g_1(c)$		g
0	→	000	→	0
1	→	001	→	1
2	→	100	→	4
3	→	010	→	2
4	→	101	→	5
5	→	110	→	6
6	→	111	→	7
7	→	011	→	3

$$\underbrace{\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&0&1&0&0 \end{bmatrix}}_{[P_{r\to t}]^T} * \underbrace{\begin{bmatrix} +&+&+&+&+&+&+&+\\ +&-&+&-&+&-&+&-\\ +&+&-&-&+&+&-&-\\ +&-&-&+&+&-&-&+\\ +&+&+&+&-&-&-&-\\ +&-&+&-&-&+&-&+\\ +&+&-&-&-&-&+&+\\ +&-&-&+&-&+&+&- \end{bmatrix}}_{[WT]} * \underbrace{\begin{bmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{bmatrix}}_{[P_{c\to g}]}$$

$$= \underbrace{\begin{bmatrix} +&+&+&+&+&+&+&+\\ +&-&+&+&-&+&-&-\\ +&+&+&-&+&-&-&-\\ +&+&-&+&-&-&-&+\\ +&-&+&-&-&-&+&+\\ +&+&-&-&-&+&+&-\\ +&-&-&-&+&+&-&+\\ +&-&-&+&+&-&+&- \end{bmatrix}}_{[MT]}$$

FIG. 4

of the feedback taps. No memory allocation is necessary for storage of the m-sequence. This greatly facilitates implementation of the cross-correlation of large arrays on microcomputers.

REFERENCES

[1] N. Zierler, *Linear recurring sequences*, J. Soc. Indust. Appl. Math., 7 (1959), pp. 31–49.
[2] S. W. Golomb, *Shift Register Sequences*, Aegean Park Press, Laguna Hills, CA, 1982.
[3] P. A. N. Briggs, P. H. Hammond, M. T. G. Hughes, and G. O. Plumb, *Correlation analysis of process dynamics using pseudo-random binary test perturbations*, Proc. Inst. Mech. Engrg., 179 (1964–65), pp. 37–50.
[4] E. P. Gyftopoulos and R. J. Hooper, *Signals for transfer-function measurements in nonlinear systems*, in Noise and Nuclear Systems, USAEC Symposium Series 4, TID-7679, United States Atomic Energy Commission, 1964, pp. 335–345.
[5] R. J. Hooper and E. P. Gyftopoulos, *On the measurement of characteristic kernels of a class of nonlinear systems*, in Neutron Noise, Waves and Pulse Propagation, USAEC Conference Report 660206, United States Atomic Energy Commission, 1967, pp. 343–356.
[6] H. R. Simpson, *Statistical properties of a class of pseudorandom sequences*, Proc. IEE-E, 103 (1966), pp. 2075–2080.
[7] N. Ream, *Nonlinear identification using inverse-repeat* m-*sequences*, Proc. IEE-E, 117 (1966), pp. 213–218.
[8] C. Swerup, *On the choice of noise for the analysis of the peripheral auditory system*, Biol. Cybernet., 29 (1978), pp. 97–104.
[9] H. A. Barker and T. Pradisthayon, *High-order autocorrelation functions of pseudorandom signals based on* m-*sequences*, Proc. IEE-E, 117 (1970), pp. 1857–1863.
[10] E. E. Sutter, *A practical non-stochastic approach to nonlinear time-domain analysis*, in Advanced Methods of Physiological Systems Modelling, Vol. 1, Biomedical Simulations Resource, Department of Biomedical Engineering, University of Southern California, Los Angeles, CA, 1987.
[11] M. Hall, Jr., *Hadamard matrices of order* 16, Lett. Propuls. Lab. Res. M., Vol. 36-10, Jet Propulsion Laboratory, Pasadena, CA, 1961, pp. 21–26.
[12] D. F. Elliott and K. R. Rao, *Fast Transforms: Algorithms, Analysis, Applications*, Academic Press, New York, 1982, p. 301.
[13] K. W. Henderson, *Comment on Computation of the fast Walsh–Fourier transform*, IEEE Trans. Comput., 19 (1970), pp. 50–51.
[14] B. J. Fino and V. R. Algazi, *Unified matrix treatment of the fast Walsh–Hadamard transform*, IEEE Trans. Comput., 25 (1976), pp. 1142–1146.
[15] H. Rademacher, *Einige Sätze von allgemeinen Orthogonalfunktionen*, Math. Ann., 87 (1922), pp. 122–138.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 695-707, August 1991
© 1991 Society for Industrial and Applied Mathematics
007

ON COUNTING LATTICE POINTS IN POLYHEDRA*

MARTIN DYER†

Abstract. Some reductions of the computational problem of counting all the integer lattice points in an arbitrary convex polyhedron in a fixed number of dimensions d are considered. It is shown that only odd d need to be studied. In three dimensions the problem is reduced to the computation of Dedekind sums. Hence it is shown that the counting problem in three or four dimensions is in polynomial time. A corresponding reduction of the five-dimensional problem is also examined, but is not shown to lead to polynomial-time algorithms.

Key words. lattice points, polynomial time, Dedekind sums, convex polyhedron

AMS(MOS) subject classifications. 52A15, 52A20, 52A25, 52A43

1. Introduction. Questions concerning the existence of integer lattice points in convex polyhedra have been well studied. The problem of determining whether the polyhedron contains *any* lattice points is the problem of integer programming. This is well known to be NP-complete in general [7], but it is an equally well-known result of Lenstra [14] that this can be done in polynomial time in any fixed dimension. (See also Kannan [10] for subsequent improvements.)

Counting *all* lattice points in a polyhedron is #P-complete, in general [24], but the status of the problem in fixed dimension is less clear. In three and four dimensions, Mordell [15] proved results concerning the numbers of lattice points in the simplex formed by cutting an orthant with a hyperplane. In the special case of pairwise coprime edge lengths (for the orthogonal sides), he established a close connection with the Dedekind sums [19]. Though his concerns were not principally computational, Mordell's paper is one of the main inspirations for the results here. Zamanskii and Cherkaskii [25], [26], [27], [28] also examined the counting problem. They were able to show that it is in polynomial time in $\mathbb{R}^2$. They analysed extensions to $\mathbb{R}^3$ but were unable to find a polynomial-time algorithm for the general three-dimensional case. Cook, Hartmann, Kannan, and McDiarmid [4] examined the problem of *approximately* counting and showed that this is polynomial-time solvable in any fixed dimension. They also showed that, in variable dimension, it is even NP-hard to approximate to within exponential factors.

There has also been interest in counting the numbers of *vertices* of the convex hull of all the lattice points in a polyhedron [23], [9], [17], [4], [2]. It has been shown that this number is bounded by a polynomial in the size of description when the dimension is fixed. This important fact is vital to the development here. Hartmann [8] describes a polynomial-time algorithm for listing all vertices of this convex hull when the dimension is fixed.

There is a wealth of related material, and the reader should note that the literature review here is in no way comprehensive, nor is it intended to be.

The main contribution of this paper is to show that there is a polynomial-time algorithm for the general lattice point counting problem for polyhedra in both three and four dimensions. The method is based on reduction to counting a particular type of simplex. This reduction is quite general. The problem of counting in even dimensions

* Received by the editors August 31, 1990; accepted for publication (in revised form) November 28, 1990.
† School of Computer Studies, University of Leeds, Leeds, United Kingdom.

is further reduced to that in lower odd dimensions. The three-dimensional problem is then shown to rest on the computation of Dedekind sums, which can be evaluated in polynomial time.

2. Definitions and notation. Throughout, $[n]=\{1,2,\cdots,n\}$, and $[m,n]=\{m,m+1,\cdots,n\}$. A *sign* means an element of $\{-1,1\}$. If $n\geqq m\geqq 0$ are integers, we write $n^{\langle m\rangle}$ for $n(n-1)\cdots(n-m+1)$ $(=1$ if $m=0)$. For $S\subseteq\mathbb{R}^d$, int S, cl S, aff S, and conv S denote the interior, closure, affine hull, and convex hull of S.

We use simplicial decompositions of (convex) polyhedra. Now any closed polyhedron has a unique partition into relatively open faces. We will call $P\subseteq\mathbb{R}^d$ a polyhedron if it is any union of relatively open faces of the closed polyhedron cl P. We write the implied relation as $P\sqsubseteq\text{cl }P$. The adjectives *open* or *closed* will be used if we wish to be more specific. We use the term *simplex* similarly. If P is a polyhedron, any face of cl P will be called a *face* of P, but it will be called *included* or *excluded*, depending on whether or not it actually belongs to P. In particular, vert P denotes the vertex set of P. We must, of course, assume that this list of open faces is supplied as part of the description of P. We observe that the maximum number of such faces is polynomial in the number of facets or vertices of P in any fixed dimension, so the list cannot be too large. A polyhedron $P\subseteq\mathbb{R}^d$ will be called *full* if int $P\neq\emptyset$. For any $S\subseteq\mathbb{R}^d$, we denote by $|S|$ the number of integer lattice points belonging to S (i.e., $|S|\stackrel{\text{def}}{=}|S\cap\mathbb{Z}^d|$). *Counting* S means evaluating $|S|$. For any convex S, the *integer hull* of S is the set $S_I=\text{conv}(S\cap\mathbb{Z}^d)$. A polyhedron P is *integral* if cl $P=P_I$, i.e., P has only integer vertices.

We use vector notation in a rather sloppy fashion. Whether a row or column vector is intended will be clear from the context. A vector may also be regarded as the ordered sequence of its coordinates or as the corresponding linked list of its coordinates. We are correspondingly sloppy about the use of the notation "dim," which simply means "dimension." Again, we believe the meaning should be clear from the context. (The multiple usage of the term "dimension" is, perhaps unfortunately, common in mathematics.)

Throughout, e_i is the ith unit vector and e a vector of all 1's. If $a\in\mathbb{R}^d$ and $\alpha\in\mathbb{R}$, the notation $a>\alpha$ (and similar) means $a>\alpha e$. We will write $a\wedge b$ for gcd (a,b). It is well known that the operation "$\wedge$" is then associative and commutative. If $x\in\mathbb{R}$, we use the (nonstandard) notation $\{\!\{x\}\!\}=x-\lfloor x\rfloor$ to denote the "fractional part." We use this principally in § 6, when considering Dedekind sums. It is traditional in this setting to use the "sawtooth" function $((x))=x-\frac{1}{2}(\lfloor x\rfloor+\lceil x\rceil)$. (See, for example, [19], [13].) This function has some nice properties for dealing with Dedekind sums and their relatives but gives no simplification of our results. Consequently, we will not use it.

3. Preliminary observations. Let P be a polyhedron such that cl $P=\{x\in\mathbb{R}^d: Ax\leqq b\}$, where A is an $m\times d$ integer matrix and b an m-vector. We consider the computational problem of determining $|P|$ when the dimension d is fixed. We call this the *d-dimensional counting problem*. Our objective is to perform the computation in polynomial time. When we use the term *polynomial* in this paper, we will usually mean polynomial in the size of the input A, b, as measured in [22]. Observe that we may equally suppose that cl P is given as a list of rational vertices, since (in fixed dimension) there is no difficulty in moving between these representations in polynomial time. Similarly, any reasonable representation of the list of included faces will suffice. We may also observe here that we lose little generality in restricting to the integer lattice since, for any lattice with rational generators, we can reduce to this case by finding a basis (in polynomial time) and then making substitutions. (See [22].)

We consider reductions of the general counting problem to that for "simpler" classes of polyhedra. Thus let us use the following terminology. If $\mathscr{C}$ is any class of polyhedra, let $\mathscr{C}_d = \{C \in \mathscr{C}: \dim(C) \leqq d\}$. If $\mathscr{D}$ is some other class, let us say that $\mathscr{C}_d$ *reduces* to $\mathscr{D}_d$ if counting any $C \in \mathscr{C}_d$ is achievable in polynomial time using an oracle that counts arbitrary $D \in \mathscr{D}_d$. We will say that $\mathscr{C}$ reduces to $\mathscr{D}$ if $\mathscr{C}_d$ reduces to $\mathscr{D}_d$ for all (fixed) d.

We use $\mathscr{A}$ to denote the class of all polyhedra. Thus if the counting problem is polynomial-time solvable in all fixed dimensions, $\mathscr{A}$ reduces to $\emptyset$. Clearly $\mathscr{A}$ reduces to the class of full polyhedra, since if P is not full, we may reduce dimension by making a suitable substitution in the inequality system. The same is true for open polyhedra, since all open faces of a polyhedron P are lower dimensional open polyhedra. Note that, in variable dimension, it is a nontrivial task [5] to determine in polynomial time the affine hull of a polyhedron in some presentations. However, in fixed dimension, this computation is clearly polynomial-time equivalent to finding the affine hull of its (explicitly presented) vertex set, a more straightforward task.

A much deeper fact is that $\mathscr{A}$ reduces to the class of open integral simplices. This may be seen as follows. First, determine the integer hull P_I of P. Because P_I has only polynomially many vertices, this can be done in polynomial time using fixed-dimensional integer programming [14], [10]. See Cook, Hartmann, Kannan, and McDiarmid [4] and Hartmann [8]. We may then triangulate P_I into a simplicial complex, such that all simplices have vertices which are also vertices of P_I. Hence we can partition P_I into open simplices of various dimensions. The conclusion now follows.

4. Reduction to a standard integral simplex. In this section we show that $\mathscr{A}$ reduces to a certain "nice" class of open simplices. For reasons discussed in § 1, it is more convenient to prove the reduction using general simplices. The proof is based on the following lemma.

LEMMA 1. *Let A be a nonsingular $d \times d$ integer matrix; then there exists a unimodular matrix U such that every element in the first row of UA is α, for some integer $\alpha \neq 0$. The matrix U can be determined in polynomial time (even when d is not fixed).*

Proof. Let $a = (\det A)eA^{-1}$. Then a is clearly an integral vector. Reduction of a to Hermite normal form (see [22, Chap. 4]) shows that there is a unimodular matrix V such that $aV = \beta e_1$, for some integer $\beta \neq 0$. Now if $U = V^{-1}$, with first row u_1, $a = \beta u_1$. Thus $u_1 = \beta^{-1}a$, and we have $u_1 A = \alpha e$, a constant vector, with $\alpha = (\det A/\beta)$. Clearly α is an integer, since u_1, A are integral. Thus U has the required property. It can be determined in polynomial time using a suitable Hermite normal form algorithm. (See [22, Chap. 5].) □

We will need the following lemma on decomposition of simplices.

LEMMA 2. *Let $S \subseteq \mathbb{R}^d$ be a full simplex with vertex set $V = \{p^0, p^1, \cdots, p^d\}$, and let $p \in \mathbb{R}^d$ be any point. Let F_i be the facet of S with* $\operatorname{vert} F_i = (V \backslash \{p^i\})$, *and let* $S_i = \operatorname{conv}(\operatorname{vert} F_i \cup \{p\})$. *Then there exist full simplices $S'_i \subseteq S_i$, $(i \in I \subseteq [0, d])$ and signs σ_i such that*

$$|S| = \sum_{i \in I} \sigma_i |S'_i|.$$

Proof. Define

$$\begin{aligned} \sigma_i &= -1 \quad \text{if aff } F_i \text{ strictly separates } p, p^i, \\ &= 0 \quad \text{if aff } F_i \text{ contains } p, \\ &= +1 \quad \text{otherwise.} \end{aligned}$$

Let $I=\{i: \sigma_i \neq 0\}$, $I^+=\{i \in I: \sigma_i=+1\}$, and $I^-=I \backslash I^+$. Then, letting $S'=\operatorname{conv}(S \cup \{p\})$, it is straightforward to show that $\operatorname{int} S_i$ $(i \in I^+)$ are the d-dimensional simplices of a simplicial complex that triangulates S', and $\operatorname{int} S_i$ $(i \in I^-)$ are the d-dimensional simplices of a similar triangulation of $S' \backslash S$. Therefore we may choose full simplices $S_i' \subseteq S_i$ $(i \in I)$ to partition S' and $S' \backslash S$. Since $|S|=|S'|-|S' \backslash S|$, the lemma follows. □

Let $\mathcal{S}$ be the class of open simplices $\{S_d(a)\}$ where, for some $a \in \mathbb{Z}^d$,

$$\operatorname{vert} S_d(a)=\left\{0, \sum_{j=1}^{k} a_j e_j \ (k \in [d])\right\}. \tag{1}$$

We will always assume, below, that $a>0$. This involves no real loss of generality since we are interested only in full simplices, and reflection in a coordinate hyperplane is a unimodular transformation.

We now observe that $S_d(a)$ has the following nice description by facets.

$$S_d(a)=\{x \in \mathbb{R}^d: 1>x_1/a_1>x_2/a_2 \cdots > x_d/a_d>0\}. \tag{2}$$

To see this, note first that all the vertices of $S_d(a)$ are in the closure of the set defined by the inequalities in (2), and only the kth vertex in (1) fails to satisfy the kth inequality as equality. (We are defining the 0th vertex in (1) to be 0, and numbering the $(d+1)$ inequalities in (2) $0, 1, \cdots, d$.)

We now prove the reduction theorem.

THEOREM 1. *Let $S \subseteq \mathbb{R}^d$ be a full integral simplex. Then there exist simplices $\Delta_1, \Delta_2, \cdots, \Delta_r$, where $r \leqq d!$, and signs σ_i $(i \in [r])$ such that*

(a) $\operatorname{int} \Delta_i \in \mathcal{S}_d$,

(b) $|S|=\sum_{i=1}^r \sigma_i |\Delta_i|$.

For fixed d, a description of $\{\Delta_i: i \in [r]\}$ can be computed in polynomial time.

Proof. We assume by induction that, for a given $t \in [0, d]$, there are full integral simplices $\Delta_1^t, \Delta_2^t, \cdots, \Delta_{r_t}^t$, with $r_t \leqq d^{\langle t \rangle}$, and corresponding signs σ_i^t $(i \in [r_t])$, such that

(a) $\operatorname{vert} \Delta_i^t$ can be ordered as $(p^0, p^1, \cdots, p^d)$, for instance, so that, for some integers α_j^i,

$$\begin{aligned} p_j^k &= \alpha_j^i \ (j \in [1, t], k \in [j, d]) \\ &= 0 \ (k \in [0, t], j \in [k+1, d]). \end{aligned}$$

(b) $|S|=\sum_{i=1}^{r_t} \sigma_i^t |\Delta_i^t|$.

The theorem is the case $t=d$ of the induction hypothesis. Since S can always be translated onto a full simplex Δ_1^0, which has its first ordered vertex at 0, the hypothesis holds for $t=0$ with $r=1$ and $\sigma_1^0=1$. Assume, then, that it holds for any $t \in [0, d-1]$. Consider a particular Δ_i^t with vertex ordering satisfying (a) of the induction hypothesis. The last $(d-t)$ coordinates of $p^0, \cdots, p^t$ are all zero, by induction, and those of $p^{t+1}, \cdots, p^d$ form the columns of a $(d-t) \times (d-t)$ integer matrix A. Singularity of this matrix would imply $\dim \Delta_i^t < d$, contradicting the assumption that Δ_i^t is full. Hence, by Lemma 1, we can determine a unimodular transformation U for A which will make its first row constant. We apply this transformation to the last $(d-t)$ coordinates of $\mathbb{R}^d$, leaving the first t invariant (i.e., we augment U by a $t \times t$ identity matrix). This transformation preserves the integer lattice, and hence leaves $|\Delta_i^t|$ unaltered. Now, however, $p^{t+1}, \cdots, p^d$ all have their $(t+1)$st component equal to α, for some integer α. Let $p=(\alpha_1^i, \alpha_2^i, \cdots, \alpha_t^i, \alpha, 0, \cdots, 0)$. We now apply Lemma 2 to (Δ_i^t, p) to conclude that Δ_i^t can be replaced by a set of full simplices $\{\Delta_j': j \in J \subseteq [0, d]\}$, where $\operatorname{vert} \Delta_j'=(\operatorname{vert} \Delta_i^t \backslash \{p^j\}) \cup \{p\}$. Thus $|J| \leqq (d+1)$, but we may bound it more tightly as follows.

Note that p has its first $(t+1)$ coordinates equal to those of $p^{t+1}, \cdots, p^d$. Thus, for $j \in [0, t]$, Δ_j' has a set of $(d-t+1)$ vertices which lie in $(t+1)$ common hyperplanes, i.e., in an affine subspace of dimension $(d-t-1)$. Thus we can find a hyperplane which includes all the vertices of Δ_j'. Hence we may assume $J \subseteq [t+1, d]$, and hence $|J| \leqq (d-t)$.

Now $\{\Delta_i^{t+1}: i \in [r_{t+1}]\}$ is formed by replacing each Δ_i^t by the set Δ_j' $(j \in J)$, derived as above. Then $r_{t+1} \leqq (d-t) r_t \leqq (d-t) d^{\langle t \rangle} = d^{\langle t+1 \rangle}$, using the induction hypothesis and the bound on $|J|$. The vertex ordering for Δ_j' may be any having $(p^0, p^1, \cdots, p^t, p)$ as an initial subsequence. Then part (a) of the induction hypothesis for the Δ_i^{t+1} is obvious from the specification of p. Part (b) follows from the final identity of Lemma 2 and (b) of the induction hypothesis for the Δ_i^t. This completes the induction. There is clearly a polynomial-time algorithm for the decomposition which directly mirrors the method of proof. □

Thus $\mathscr{A}$ reduces to $\{P: \text{int } P \in \mathscr{S}\}$. Now $\mathscr{A}$ will reduce to $\mathscr{S}$ immediately if $\mathscr{S}$ is closed under the operation of taking subfaces. We prove this next. Let p^i $(i \in [0, d])$ be the ith ordered vertex of $S_d(a)$. Let F be any face of $S_d(a)$, with vert $F = \{p^i: i \in I_F\}$. Consider the following procedure applied to the d-vector a, viewed as a formal list.

function $b(a, F)$
(1) **for** $i \in [d-1]$ **do**
  if $i \notin I_F$ **then** insert the (g.c.d.) operation $\wedge$ between a_i and a_{i+1}.
(2) Evaluate all the $\wedge$ operations to give the reduced vector b, for instance.
(3) **if** $0 \notin I_F$ **then** delete the first element of b.
  if $d \notin I_F$ **then** delete the last element of b.
(4) $b(a, F) \leftarrow b$.

Clearly $b(a, F)$ is a vector with dim $b = \dim F$. Call $b = b(a, F)$ a *face-vector* of a. Clearly any face-vector can be obtained in polynomial time. Now we have the following lemma.

LEMMA 3. *If F is an open face of $S_d(a)$ with* $\dim F = k$, *then* $|F| = |S_k(b)|$, *where* $b = b(a, F)$.

Proof. Since the g.c.d. operator is associative, it is clearly sufficient to prove this for F a facet and to use induction. If $0 \notin I_F$, then we must have $x_1/a_1 = 1$ on F, and the lemma follows directly. Similarly if $d \notin I_F$, we have $x_d/a_d = 0$. If $i \notin I_F$ $(i \in [d-1])$, we have $x_i/a_i = x_{i+1}/a_{i+1}$ on F. Let $\lambda = a_i \wedge a_{i+1}$, $\alpha = a_i/\lambda$, $\beta = a_{i+1}/\lambda$. It follows from simple divisibility considerations that we must have $x_i = x'\alpha$, $x_{i+1} = x'\beta$ for $x' \in [\lambda - 1]$ at integer points on F. Thus $x_i/a_i = x_{i+1}/a_{i+1} = x'/\lambda$ at all such points. The lemma now follows. □

COROLLARY 1. *$\mathscr{A}$ reduces to $\mathscr{S}$.*

Remark 1. The simplices, $\mathscr{M} = \{M_d(a)\}$, considered by Mordell [15], were

$$M_d(a) = \left\{ x \in \mathbb{R}^d: \sum_{j=1}^{d} x_j/a_j < 1,\ x_1/a_1 > 0,\ x_2/a_2 > 0, \cdots, x_d/a_d > 0 \right\},$$

so vert $M_d(a) = \text{conv}\{0, a_j e_j\ (k \in [d])\}$. For $d \leqq 2$, $\mathscr{M}_d$ and $\mathscr{S}_d$ are essentially the same, but this is not true for $d \geqq 3$. The class $\mathscr{M}$ may appear simpler than $\mathscr{S}$, but we do not know whether $\mathscr{A}$ reduces to $\mathscr{M}$.

5. Even dimensions. The main result of this section is to show that, if d is even, $\mathscr{A}_d$ reduces to $\mathscr{A}_{d-1}$.

If $\gamma \in [3]^{d-1}$, consider the following algorithm applied to $a \in \mathbb{Z}_+^d$.

function $\phi(\gamma, a)$
(1) **for** $i \in [d-1]$ **do**

if $\gamma_i = 1$ **then** split the list between a_i and a_{i+1}.
if $\gamma_i = 2$, **then** insert the operation $\wedge$ between a_i and a_{i+1}.

(2) Evaluate all the g.c.d. operations in the sublists.
Let b_j ($j \in [r]$) be the resulting reduced sublists (i.e., vectors).
Let $k_j = \dim b_j$ and $t = |\{i: \gamma_i = 1\}|$.

(3) $\phi(\gamma, a) \leftarrow (-1)^{d-1-t} \prod_{j=1}^{r} |S_{k_j}(b_j)|$.

Then we have the following theorem.

THEOREM 2. *Let d be even, and let $\Gamma = [3]^{d-1} \backslash \{3e\}$. Then*

$$|S_d(a)| = \frac{1}{2} \sum_{\gamma \in \Gamma} \phi(\gamma, a).$$

Proof. The method is "inclusion-exclusion," using a natural symmetry of $S_d(a)$. Let

$$R_d(a) = [a_1 - 1] \times [a_2 - 1] \times \cdots \times [a_j - 1].$$

If $x \in R_d(a)$ (which we will abbreviate to R) then, from (2), under the bijection $x \mapsto (a - x)$ on R,

$$|S_d(a)| = |\{x \in R: x_1/a_1 < x_2/a_2 < \cdots < x_d/a_d\}|. \tag{3}$$

For $i \in [d-1]$, let us write $\lambda_i(x) = (x_i/a_i - x_{i+1}/a_{i+1})$. For $\rho \in \{<, =, >\}$, let δ_i^ρ be the indicator function of $\lambda_i(x)\ \rho\ 0$. Then, from (2) and (3),

$$|S_d(a)| = \sum_{x \in R} \prod_{i=1}^{d-1} \delta_i^{>}(x) = \sum_{x \in R} \prod_{i=1}^{d-1} \delta_i^{<}(x). \tag{4}$$

However, we have $\delta_i^{<}(x) = 1 - \delta_i^{>}(x) - \delta_i^{=}(x)$. Thus the last expression of (4) implies

$$|S_d(a)| = \sum_{x \in R} \prod_{i=1}^{d-1} (1 - \delta_i^{=}(x) - \delta_i^{>}(x)). \tag{5}$$

Expanding the product in (5), we obtain $|S_d(a)|$ as the sum of 3^{d-1} terms, each of the form

$$\sigma \sum_{y \in R} \prod_{i=1}^{d-1} \zeta_i(y), \tag{6}$$

where σ is a sign, and $\zeta_i \in \{\delta_i^{>}, \delta_i^{=}, 1\}$. Each ζ_i is an indicator function, so the product in (6) is the indicator of an intersection of sets. For each i there are three possibilities. The case $\zeta_i = 1$ is equivalent to deleting the $(i+1)$st inequality in (2), so the inequality system "decomposes" on R into

$$\{x_1/a_1 > \cdots > x_i/a_i\} \times \{x_{i+1}/a_{i+1} > \cdots > x_d/a_d\}.$$

This corresponds to "splitting" the vector a, i.e., to $\gamma_i = 1$ in the computation of $\phi(\gamma, a)$. Having $\zeta_i = \delta_i^{=}$ corresponds to replacing the $(i+1)$st inequality in (2) by an equality. This is equivalent to inserting the g.c.d. operation in a, i.e., to $\gamma_i = 2$ in the computation of $\phi(\gamma, a)$ (cf. Lemma 3). Finally, $\zeta_i = \delta_i^{>}$ corresponds to imposing the $(i+1)$st inequality in (2), i.e., to $\gamma_i = 3$ in the computation of $\phi(\gamma, a)$. The sign σ is clearly $(-1)^{d-1-t}$, where t is the number of i for which $\zeta_i = 1$. This corresponds to the sign in the computation of $\phi(\gamma, a)$. Thus each of the 3^{d-1} sums is equal to a unique $\phi(\gamma, a)$. However, using (4), we have $\phi(3e, a) = (-1)^{d-1}|S_d(a)| = -|S_d(a)|$, since d is even. The theorem now follows from (5). $\square$

THEOREM 3. *$\mathscr{A}$ reduces to $\mathscr{S}^o = \{S_d(a) \in \mathscr{S}\ (d\ odd)\}$.*

Proof. For even d, the computation of $\phi(\gamma, a)$ for $\gamma \neq 3e$ involves only determination of $|S_k(b)|$ for $k<d$, and b a face-vector of a with $\dim b = k$. The result follows by induction on d. □

COROLLARY 2. $S_1(p) = p-1$, $|S_2(p,q)| = \frac{1}{2}((p-1)(q-1)-(p \wedge q - 1))$.

Proof. The first assertion is obvious. The second follows from this and Theorem 2. □

COROLLARY 3.

$$\sum_{x=1}^{p-1} \lfloor qx/p \rfloor = \tfrac{1}{2}((p-1)(q-1)+(p \wedge q - 1)),$$

$$\sum_{x=1}^{p-1} \lceil qx/p \rceil = \tfrac{1}{2}(pq + p - q - p \wedge q).$$

Proof. Using Lemma 3,

$$\sum_{x=1}^{p-1} \lfloor qx/p \rfloor = |\{0 < y/q \leqq x/p < 1\}| = |S_2(q,p)| + |S_1(p \wedge q)|,$$

$$\sum_{x=1}^{p-1} \lceil qx/p \rceil = |\{0 \leqq y/q < x/p < 1\}| = |S_2(q,p)| + |S_1(p)|.$$

The results now follow from these and Corollary 2. □

COROLLARY 4. *Two-dimensional counting can be done in polynomial time.*

Proof. The proof follows from Corollary 2 and Theorem 3. □

Remark 2. The result of Corollary 4 was previously obtained, using different methods, by Zamanskii and Cherkaskii (see [25]-[28]).

Remark 3. For any suitably defined convex body K in $\mathbb{R}^d$ the feasibility question $|K| \overset{?}{\geq} 0$ can be answered in polynomial time by fixed-dimensional integer programming. (See Kannan [10, p. 434]. By "suitably defined" here, we mean "given by a (polynomial time) separation oracle.") Hence the integer hull K_I of such a body could be determined in polynomial time by the "gift wrapping" idea (see [18, p. 125]) *provided* K_I has only polynomially many vertices. Since K_I is a polyhedron, counting K could then be achieved as above. We might therefore hope that Corollary 4 would generalise. However, it appears to fail for very simple convex sets in $\mathbb{R}^2$. To see this, consider

$$K(n) = \{(x,y) \in \mathbb{R}^2 \colon xy \geqq n, 1 \leqq (x,y) \leqq n\}.$$

It is easy to see that $|K(n+1)| - |K(n)| = 2n-1$ if and only if n is prime. A polynomial-time algorithm for counting $K(n)$ therefore implies a similar algorithm for primality testing. Thus we might guess that $K_I(n)$ can have nonpolynomially many vertices. It is easy to see that this can happen. Let $m \geqq 114$, and n be the product of the first $\lceil m/\ln m \rceil$ primes. Each prime is at most m by a form of the Prime Number Theorem [21]. Thus $n < 3^m$, say, so m measures the input size of $K(n)$. However, n has at least $2^{m/\ln m}$ (ordered) two-term factorizations, i.e., nonpolynomially many. Each gives a vertex of $K_I(n)$. Thus the approach to counting $K(n)$ used here is doomed at the outset. (See Remark 8 below for an even worse example.)

Remark 4. The proof of Theorem 2 leads to a closed formula for $S_d(a)$. Note that $-1 < \lambda_i(x) < 1$ on R, and thus $\delta_i^{>}(x) = \lceil \lambda_i(x) \rceil$. Thus, from (4),

$$|S_d(a)| = \sum_{x_1=1}^{a_1-1} \sum_{x_2=1}^{a_2-1} \cdots \sum_{x_d=1}^{a_d-1} \left\lceil \frac{x_1}{a_1} - \frac{x_2}{a_2} \right\rceil \left\lceil \frac{x_2}{a_2} - \frac{x_1}{a_1} \right\rceil \cdots \left\lceil \frac{x_d}{a_d} - \frac{x_{d-1}}{a_{d-1}} \right\rceil.$$

Unfortunately, this expression is not directly computable in polynomial time.

We prove one further general reduction, that the elements of a need have no common divisor. We place it here since it has some superficial similarities to Theorem 2. For this, it is convenient to use

$$S'_d(a)=\{1>x_d/a_d>\cdots>x_1/a_1\geqq 0\},$$

rather than $S_d(a)$. Let $\gamma\in[2]^{d-1}$, $a\in\mathbb{Z}_+^d$, and $\lambda\in\mathbb{Z}_+$. Consider

function $\zeta(\gamma, a, \lambda)$
(1) **for** $i\in[d-1]$ **do**
  if $\gamma_i=1$ **then** split the list between a_i and a_{i+1}.
(2) Let b_j $(j\in[r])$ be the resulting sublists of a, $k_j=\dim b_j$.
(3) $\zeta(\gamma, a, \lambda)\leftarrow\binom{\lambda}{r}\prod_{j=1}^{r}|S'_{k_j}(b_j)|$.

Then we have the following lemma.

LEMMA 4. $|S'_d(\lambda a)|=\sum_{\gamma\in[2]^{d-1}}\zeta(\gamma, a, \lambda)$.

Proof. Partition the $x\in S'_d(\lambda a)$ into *boxes* according to $\lfloor x_i/a_i\rfloor=s_i-1$. A box corresponds to a nonincreasing sequence $s=(s_1,\cdots,s_d)$ such that $s_i\in[\lambda]$. Any such s splits into maximal subsequences for which s_i has the same value. Suppose there are r distinct s_i. There are exactly $\binom{\lambda}{r}$ ways of choosing these distinct values. For each choice, the possible s can then be formed by splitting a d-sequence into r nonempty parts, and then assigning the r values, in decreasing order, to the successive parts. Any split into r parts corresponds to choosing a γ. For a given split, suppose $s_j=\cdots=s_{j+k-1}$ $(=\xi-1)$ is any part, and let $b=(a_j,\cdots,a_{j+k-1})$ be the corresponding part of a. In $S'_d(a)$, we must have

$$(\xi+1)>x_j/a_j>\cdots>x_{j+k-1}/a_{j+k-1}\geqq\xi.$$

But this set has a bijection $x_l\mapsto(x_{j+l-1}-\xi a_{j+l-1})$ $(l\in[k])$ with $S'_k(b)$. The lemma now follows. □

Remark 5. This lemma is closely related to the theorem that the number of lattice points in a polyhedron varies polynomially under the operation of subdivision of the lattice. (See, for example, [16].) Unfortunately, it does not seem that we can apply this result directly to get our conclusion here.

THEOREM 4. *$\mathscr{A}$ reduces to $\mathscr{S}^*=\{S_d(a)\in\mathscr{S}^o\colon a_1\wedge\cdots\wedge a_d=1\}$.*

Proof. By induction on d, the result follows from Theorem 3, Lemma 4, and the equation $|S'_d(a)|=|S_d(a)|+|S_{d-1}(a')|$ (where $a'=(a_1,\cdots,a_{d-1})$), which follows from Lemma 3. □

6. Dedekind sums and three dimensions. From Theorem 4, three-dimensional counting clearly reduces to counting $S_3(r,p,q)$, where $p\wedge q\wedge r=1$. Then, however, using Lemma 3 and Corollary 2,

$$|S_3(r,p,q)|=N-|S_2(r,p\wedge q)|=N-\tfrac{1}{2}(p\wedge q-1)(r-1),$$

where, if (z,x,y) is the typical point of $\mathbb{R}^3$,

$$N=|\{0<y/q\leqq x/p<z/r<1\}|. \tag{7}$$

It clearly suffices to determine N. But since, for any integer $0<x<p$, there are $\lfloor qx/p\rfloor$ values of y, and $(r-1-\lfloor rx/p\rfloor)$ values of z in (7),

$$\begin{aligned} N&=\sum_{x=1}^{p-1}\lfloor qx/p\rfloor(r-1-\lfloor rx/p\rfloor),\\ &=(r-1)\sum_{x=1}^{p-1}\lfloor qx/p\rfloor-\sum_{x=1}^{p-1}\lfloor qx/p\rfloor\lfloor rx/p\rfloor, \qquad (8)\\ &=\frac{1}{2}(r-1)((p-1)(q-1)-(p\wedge q-1))-\sum_{x=1}^{p-1}\lfloor qx/p\rfloor\lfloor rx/p\rfloor,\end{aligned}$$

using Corollary 2. It thus suffices to determine $\sum_{x=1}^{p-1} \lfloor qx/p \rfloor \lfloor rx/p \rfloor$. Now let

$$D_p^{q,r} \stackrel{\text{def}}{=} \sum_{x=1}^{p-1} \{\!\{qx/p\}\!\}\{\!\{rx/p\}\!\},$$

$$D_p^q \stackrel{\text{def}}{=} D_p^{q,1} = \sum_{x=1}^{p-1} (x/p)\{\!\{qx/p\}\!\}, \tag{9}$$

$$D_p \stackrel{\text{def}}{=} D_p^1 = \sum_{x=1}^{p-1} (x/p)^2 = (p-1)(2p-1)/6p.$$

The order of superscipts in $D_p^{q,r}$ is clearly immaterial. Now, using the above notation,

$$\sum_{x=1}^{p-1} \lfloor qx/p \rfloor \lfloor rx/p \rfloor = \sum_{x=1}^{p-1} (qx/p - \{\!\{qx/p\}\!\})(rx/p - \{\!\{rx/p\}\!\})$$
$$= rqD_p - qD_p^r - rD_p^q + D_p^{q,r}.$$

Thus we have only to evaluate the sum $D_p^{q,r}$ in polynomial time in order to have a polynomial-time algorithm for three-dimensional counting. Since $D_p^{q,r}$ has $(p-1)$ terms, it is not obvious that this is possible. However, sums of this type have been well studied, since the D_p^q are (essentially) the "Dedekind sums" [19]. We first show that $D_p^{q,r}$ can be determined using only a polynomial-time algorithm for evaluating D_p^q in the special case $p \wedge q = 1$. We need the following simple lemma. This lemma is well known in relation to Dedekind sums, as is some of the other content of this section, but we give proofs, since they are all fairly short.

LEMMA 5. *If* $\theta \in \mathbb{R}$ *and* $p \wedge q = 1$, *then* $\sum_{x=0}^{p-1} \{\!\{(qx+\theta)/p\}\!\} = \{\!\{\theta\}\!\} + \frac{1}{2}(p-1)$.

Proof. Substitute $x \mapsto q^{-1}x \bmod p$ into the sum. (Because $p \wedge q = 1$, q^{-1} exists and the mapping is bijective.) The sum is then $\sum_{x=0}^{p-1} \{\!\{(x+\theta)/p\}\!\}$. With a further variable change $x \mapsto (x - \lfloor \theta \rfloor) \bmod p$, this is $\sum_{x=0}^{p-1} (x + \{\!\{\theta\}\!\})/p$, giving the result. □

We now prove the claimed reduction.

LEMMA 6. *If* $p \wedge q \wedge r = 1$, $\lambda = p \wedge q$, $(\alpha, \beta) = (p, q)/\lambda$, *then*

$$D_p^{q,r} = D_\alpha^{\beta,r} + \tfrac{1}{4}(\lambda - 1)(\alpha - 1).$$

Proof. Putting $x = \mu\alpha + \nu$ ($\mu \in [0, \lambda - 1]$, $\nu \in [0, \alpha - 1]$),

$$D_p^{q,r} = \sum_{\nu=0}^{\alpha-1} \sum_{\mu=0}^{\lambda-1} \{\!\{(r\mu + r\nu/\alpha)/\lambda\}\!\}\{\!\{\beta\nu/\alpha\}\!\}.$$

Using Lemma 5 on the inner sum gives

$$D_p^{q,r} = \sum_{\nu=0}^{\alpha-1} \left(\{\!\{r\nu/\alpha\}\!\} + \frac{1}{2}(\lambda - 1) \right) \{\!\{\beta\nu/\alpha\}\!\}.$$

Applying Lemma 5 again (with $\theta = 0$) on the second term in this sum gives the conclusion. □

Therefore we may suppose that $p \wedge q = 1$.

Remark 6. The assumption $p \wedge q \wedge r = 1$ is not entirely necessary, since if $\lambda = p \wedge q \wedge r$ and $(\alpha, \beta, \gamma) = (p, q, r)/\lambda$, we can easily show that $D_p^{q,r} = \lambda D_\alpha^{\beta,\gamma}$.

LEMMA 7. *If* $p \wedge q = 1$, *then* $D_p^{q,r} = D_p^t$, *where* $t \equiv rq^{-1} \pmod{p}$.

Proof. Change variable $x \mapsto q^{-1}x \bmod p$ (cf. proof of Lemma 5). □

We have thus reduced to evaluating D_p^q. But, by Lemma 6, we may assume $p \wedge q = 1$ are coprime. All the work required so far can be done in polynomial time using only

the Euclidean algorithm. It remains only to show how to evaluate D_p^q in polynomial time in the case $p \wedge q = 1$. That this can be done is a direct consequence of the famous "reciprocity relation" of Dedekind. Many proofs of this identity, and generalizations, are known (see [19], [13]). Since we have the machinery available, we give a short proof for completeness.

LEMMA 8. *If* $p \wedge q = 1$, *then*

$$D_p^q + D_q^p = \tfrac{1}{4}(p+q-3) + \tfrac{1}{12}(p/q + q/p + 1/pq).$$

Proof. The variable change $x \mapsto q^{-1}x \bmod p$ implies $D_p^{q,q} = D_p$. Thus

$$\sum_{x=1}^{p-1} \lfloor qx/p \rfloor^2 = \sum_{x=1}^{p-1} (qx/p - \{\!\{qx/p\}\!\})^2 = (q^2+1)D_p - 2qD_p^q. \tag{10}$$

Since qx/p, py/q are not integral for $x \in [p-1]$, $y \in [q-1]$,

$$\begin{aligned} \sum_{x=1}^{p-1} \lfloor qx/p \rfloor^2 &= \sum_{0<y/q<x/p<1} (2y-1) \\ &= \sum_{0<x/p<y/q<1} (2(q-y)-1) \\ &= \sum_{y=1}^{q-1} ((2q-1)-2y)\lfloor py/q \rfloor, \\ &= \frac{1}{2}(2q-1)(p-1)(q-1) - 2\sum_{y=1}^{q-1} y(py/q - \{\!\{py/q\}\!\}) \\ &= (p-3)qD_q - 2qD_q^p, \end{aligned} \tag{11}$$

where the first line involves an elementary sum, the second follows by making the variable change $x \mapsto (p-x)$, $y \mapsto (q-y)$, the fourth by using Corollary 3, and the fifth by using (9). The lemma now follows by equating (10) and (11), using (9), and simplifying. □

COROLLARY 5. *$D_p^{q,r}$ can be evaluated in polynomial time.*

Proof. We need only consider D_p^q with $p \wedge q = 1$. If $q > p$, then clearly $D_p^q = D_p^{q'}$, where $q' = q \bmod p$. This, with Lemma 8, implies an algorithm whose behaviour and analysis closely parallel those of the Euclidean algorithm. (Note, since $p \wedge q = 1$, we finally reach $D_1^p = 0$.) □

Remark 7. Lemma 8 is clearly elementary, but was discovered by Dedekind in the context of modular function theory. (See [1, Chap. 3] for an introduction.) The book by Rademacher and Grosswald [19] is an exhaustive account of the known facts on Dedekind sums at the time of publication (1972). The algorithm for the calculation of the sums was probably known from Dedekind onwards, as was its "computational efficiency." More recently, explicitly algorithmic treatments have been given, for example, by Knuth [12], [13].

In consequence of the results of this section, we have Theorem 5.

THEOREM 5. *Three-dimensional counting can be done in polynomial time.* □

From Theorem 3, we can therefore conclude with Theorem 6.

THEOREM 6. *Four-dimensional counting can be done in polynomial time.* □

Remark 8. In $\mathbb{R}^4$, counting more general convex bodies appears even harder than was implied for $\mathbb{R}^2$ by Remark 3. The following observation is due to Kannan [11]. Let

$$B_4(n) = \{x \in \mathbb{R}^4 \colon x_1^2 + x_2^2 + x_3^2 + x_4^2 \leqq n\}.$$

It is a classical result of Jacobi (see, for example, [6]) that

$$r_4(n) = |B_4(n)| - |B_4(n-1)| = 8 \sum_{4 \nmid m | n} m.$$

Thus if n is the product of distinct odd primes p, q, then $r_4(n)=8(1+p+q+n)$. This, together with $n=pq$, is sufficient to determine p and q. Therefore (this type of) factorization can be done in polynomial time given a counting oracle for $B_4(n)$. In particular, a polynomial-time algorithm for counting $B_4(n)$ would imply a similar algorithm for breaking the RSA cryptosystem [20]. All $r_4(n)$ points are vertices of the integer hull of $B_4(n)$, i.e., "exponentially" many.

7. Beyond four dimensions. It is not clear how to extend the result of § 6 to higher dimensions, since we have no polynomial-time algorithm for evaluating sums analogous to the $D_p^{q,r}$, i.e., of the form

$$D_p^{q_1,\cdots,q_{d-1}}=\sum_{x=1}^{p-1}\{\!\{q_1x/p\}\!\}\{\!\{q_2x/p\}\!\}\cdots\{\!\{q_{d-1}x/p\}\!\}. \tag{12}$$

It is even unclear to what extent the problem can be reduced to the computation of such sums. The reader may check that such sums are sufficient if and only if $\mathscr{A}$ reduces to the class of d-pyramids $\mathscr{P}=\{P_d(a)\colon d \text{ odd}\}$, where

$$P_d(a)=\{0<x_i/a_i<x_d/a_d<1\ (i\in[d-1])\}.$$

We are unable to prove this in general. However, we are able to show that $\mathscr{A}_6$ reduces to $\mathscr{P}_5$. To establish this, it clearly suffices to show that $\mathscr{S}_5$ reduces to $\mathscr{P}_5$. We will briefly outline this reduction below, leaving the interested reader to supply some of the details. Note that the converse implication is true, however. If we could solve the counting problem in d dimensions, then, by counting polyhedra in the class $P_d(a)$, we could certainly evaluate sums of the form (12).

For $a\in\mathbb{Z}_+^5$, let $a^i=(a_1,\cdots,a_i)$ $(i\in[5])$, and, for given a^i,

$$f_i(y)=|\{1>x_1/a_1>\cdots>x_{i-1}/a_{i-1}>y/a_i\}|.$$

Let

$$Y(a)=\{1>x_1/a_1>x_2/a_2>x_3/a_3>(x_4/a_4,x_5/a_5)>0\},$$
$$X(a)=\{1>(x_1/a_1,x_2/a_2)>x_3/a_3>(x_4/a_4,x_5/a_5)>0\}.$$

We first reduce $S_5(a)$ to $Y(a)$, then to $X(a)$. Simple counting gives

$$|S_5(a)|+|S_5(a^3,a_5,a_4)|=|Y(a)|-|S_4(a^3,a_4\wedge a_5)|. \tag{13}$$

Letting $g(x)=(\lceil x\rceil-1)$, we can also show easily that

$$\begin{aligned}|S_5(a)|&=\sum_{y=1}^{a_4-1}g(a_5y/a_4)f_4(y)\\&=\lfloor a_5/a_4\rfloor\sum_{y=1}^{a_4-1}yf_4(y)+\sum_{y=1}^{a_4-1}g(\{\!\{a_5/a_4\}\!\}y)f_4(y)\\&=\lfloor a_5/a_4\rfloor(|S_5(a^4,a_4)|+|S_4(a^4)|)+|S_5(a^4,a_5\bmod a_4)|,\end{aligned} \tag{14}$$

$$\begin{aligned}|S_5(a^4,a_4)|&=\sum_{y=1}^{a_3-1}\frac{1}{2}g(a_4y/a_3)(g(a_4y/a_3)+1)f_3(y)\\&=\frac{1}{2}|Y(a^4,a_4)|+\frac{1}{2}|S_4(a^4)|.\end{aligned} \tag{15}$$

From (13), (14), and (15) we can construct a "Euclidean" algorithm which reduces counting $S_5(a)$ to counting a polynomial number of Y's. Essentially the same method,

after using the bijection $x \mapsto (a-x)$, reduces counting $Y(a)$ to counting a polynomial number of X's. Thus we need only to count $X(a)$. But $|X(a)|$ can be expressed as a single sum over x_3, by a similar argument to that leading from (7) to (8). This sum can then be manipulated into the required form. With a little further work, we can show the following lemma.

LEMMA 9. *Five-dimensional counting is polynomial-time (Turing) equivalent to computing the sums*

$$D_p^{q,r,s,t} = \sum_{x=1}^{p-1} \{\!\{qx/p\}\!\}\{\!\{rx/p\}\!\}\{\!\{sx/p\}\!\}\{\!\{tx/p\}\!\},$$

where $q \mid p$ *and* $q \wedge r \wedge s \wedge t = 1$.

Remark 9. The difficulty of computing these "generalized" Dedekind sums (i.e., sums like (12) with odd $d \geqq 5$) is that the "reciprocity relations" which can be obtained (analogously to Lemma 8) are in terms of three or more such sums. (See, for example, [3].) The "Euclidean algorithm" approach therefore leads to branching (and nonpolynomial behaviour) when the number of "parameters" is greater than two. Thus, it is not clear whether these reciprocity relationships are actually useful from a computational viewpoint. (See, for example, the pitfall in the main idea of [28].)

8. Concluding remarks. By reducing to Dedekind sums, we have shown that counting in up to four dimensions can be done in polynomial time. We have been somewhat vague about the complexity of the algorithm, but the reader may check (using [4], [12], [22] for the necessary estimates) that the running time is dominated by the $O((m\phi)^{2d})$ time needed to determine the integer hull P_I. (Here $d = 2, 3$, or 4 is the dimension of P, m is the number of inequalities in the system defining P, and ϕ is the maximum size of any inequality. See [22].)

Obviously, the major question left unresolved is whether a similar result holds in five dimensions (and hence six). A polynomial-time algorithm for evaluating the sums of Lemma 9 would, of course, settle the counting problem for six dimensions. More generally, we might hope that the corresponding result is true for any fixed number of dimensions, as with integer programming. We conjecture that this is the case, though a solution seems to require some new techniques.

Of course, it may be that d-dimensional counting is not in polynomial time for some $d > 4$. Proving #P-completeness, or even NP-hardness, seems likely to be extremely difficult (even if true). It might be possible to reduce some difficult number theoretic problem like factorization to counting, as was done for $B_4(n)$ in Remark 8. However, this also appears tricky, since there is no apparent relationship between linear inequalities and nonlinear problems like factorization.

A less ambitious aim is to establish whether $\mathscr{A}$ reduces to any "nice" classes of polyhedra other than $\mathscr{S}$, for example, the pyramids $\mathscr{P}$ or the Mordell simplices $\mathscr{M}$. Reduction to $\mathscr{P}$ would be interesting, since it would imply the equivalence of counting to evaluating sums like (12).

Acknowledgments. I am indebted to Ravi Kannan on several counts: for bringing the problem to my attention, for providing several key references, and for many informative discussions. I am grateful also to Alan Frieze and David Applegate for useful conversations.

REFERENCES

[1] T. M. APOSTOL, *Modular Functions and Dirichlet Series in Number Theory*, Springer-Verlag, New York, 1976.

[2] I. BÁRÁNY, R. HOWE, AND L. LOVÁSZ, *On integer points in polyhedra: A lower bound*, Combinatorica, to appear.

[3] L. CARLITZ, *A note on generalized Dedekind sums*, Duke J. Math., 21 (1954) pp. 399-403.

[4] W. COOK, M. HARTMANN, R. KANNAN, AND C. MCDIARMID, *On integer points in polyhedra*, Combinatorica, submitted.

[5] J. EDMONDS, L. LOVÁSZ, AND W. R. PULLEYBLANK, *Brick decompositions and the matching rank of graphs*, Combinatorica, 2 (1982), pp. 247-274.

[6] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fourth Edition, Oxford University Press, Oxford, 1960.

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman, San Francisco, CA, 1979.

[8] M. HARTMANN, *Cutting planes and the complexity of the integer hull*, Ph.D. thesis, Cornell University, Ithaca, NY, 1989.

[9] A. C. HAYES AND D. G. LARMAN, *The vertices of the knapsack polytope*, Discrete Appl. Math., 6 (1983), pp. 135-138.

[10] R. KANNAN, *Minkowski's convex body theorem and integer programming*, Math. Oper. Res., 12 (1987), pp. 415-440.

[11] ———, personal communication.

[12] D. E. KNUTH, *The Art of Computer Programming Vol.* II: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1973.

[13] ———, *Notes on generalized Dedekind sums*, Acta Arithmetica, 23 (1977), pp. 297-325.

[14] H. W. LENSTRA, *Integer programming with a fixed number of variables*, Math. Oper. Res., 8 (1983), pp. 538-548.

[15] L. J. MORDELL, *Lattice points in a tetrahedron and generalized Dedekind sums*, J. Indian Math. Soc., 15 (1951), pp. 41-46.

[16] I. G. MCDONALD, *The volume of a lattice polyhedron*, Proc. Cambridge Philos. Soc., 59 (1963), pp. 719-726.

[17] D. A. MORGAN, *Upper and lower bound results on the convex hull of integer points in polyhedra*, Mathematika, submitted.

[18] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, New York, 1985.

[19] H. RADEMACHER AND E. GROSSWALD, *Dedekind sums*, Math. Assoc. Amer. Carus Monograph, No. 16, 1972.

[20] R. RIVEST, A. SHAMIR, AND L. ADLEMAN, *A method for obtaining digital signatures and public key cryptosystems*, Comm. ACM, 21 (1978), pp. 120-126.

[21] J. B. ROSSER AND L. SHOENFELD, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 66-94.

[22] A. SCHRIJVER, *The Theory of Linear and Integer Programming*, John Wiley, Chichester, U.K., 1986.

[23] V. N. SHEVCHENKO, *On the number of extreme points in integer programming*, Kibernetika, 2 (1981), pp. 133-134.

[24] L. G. VALIANT, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410-421.

[25] L. YA. ZAMANSKII AND V. L. CHERKASKII, *Determination of the number of integer points in polyhedra in $\mathbb{R}^3$: Polynomial algorithms*, Dokl. Akad. Nauk. Ukrain. USSR Ser. A 4 (1983), pp. 13-15.

[26] ———, *A formula for finding the number of integer points under a line and an application*, Ekonomika i Mat. Metody, 20 6 (1984), pp. 1132-1138.

[27] ———, *Effective algorithms for the solution of discrete optimization problems*, Znanie, Kiev, USSR, 1984.

[28] ———, *Generalization of the Jacobi-Perron algorithm for determining the number of integer points in polyhedra*, Dokl. Akad. Nauk. Ukrain. USSR Ser. A 10 (1985), pp. 10-13.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 708–725, August 1991

© 1991 Society for Industrial and Applied Mathematics

008

PARALLEL TRANSITIVE CLOSURE AND POINT LOCATION IN PLANAR STRUCTURES*

ROBERTO TAMASSIA† AND JEFFREY S. VITTER†

Abstract. Parallel algorithms for several graph and geometric problems are presented, including transitive closure and topological sorting in planar *st*-graphs, preprocessing planar subdivisions for point location queries, and construction of visibility representations and drawings of planar graphs. Most of these algorithms achieve optimal $O(\log n)$ running time using $n/\log n$ processors in the EREW PRAM model, n being the number of vertices.

Key words. parallel algorithms, parallel computation, graph algorithms, planar *st*-graphs, transitive closure, reachability, planar point location, computational geometry, fractional cascading, graph drawing, visibility

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction. Planar *st*-graphs, which include series-parallel graphs as a special case, were first introduced by Lempel, Even, and Cederbaum [34] in connection with a planarity testing algorithm, and they have subsequently been used in a host of applications, dealing with partial orders [30]; planar graph embedding [6], [14], [49]; graph planarization [37]; graph drawing [13], [15]; floor planning [57]; planar point location [19], [39]; visibility [36], [42], [52], [54], [58], [59]; motion planning [41]; and VLSI layout compaction [57].

In this paper, we present a new technique for constructing in parallel an implicit representation of the transitive closure of a planar *st*-graph. This technique is further applied to obtain optimal parallel algorithms for the following problems:

(1) transitive closure, reachability, and topological sorting in planar *st*-graphs;
(2) preprocessing planar subdivisions for point location queries;
(3) construction of visibility representations and drawings of planar graphs.

We adopt the standard *parallel random-access machine* (PRAM) model of computation, in which processors concurrently access a shared memory [29] . Communication costs are not taken into account by this model; the time to access a memory location is constant for each processor. An exclusive-read exclusive-write (EREW) PRAM prohibits concurrent access to the same location of the shared memory. A concurrent-read exclusive-write (CREW) PRAM allows concurrency for reads but not for writes. A concurrent-read concurrent-write (CRCW) PRAM allows concurrent reading and concurrent writing, under various conventions for concurrent writing. Our algorithms use the most restrictive EREW PRAM.

Computing the transitive closure of a digraph G with n vertices can be done sequentially in linear time, but the best known parallel algorithms require $O(\log^2 n)$

* Received by the editors November 14, 1989; accepted for publication (in revised form) July 17, 1990. An extended abstract of this paper was presented at the 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, New Mexico, June 1989, and at the International Workshop on Discrete Algorithms and Complexity, Fukuoka, Japan, November 1989.

† Department of Computer Science, Brown University, Providence, Rhode Island 02912–1910.
This research was supported in part by grant CCR-9007851 from the National Science Foundation and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146, ARPA order 6320, Amendment 1, and by grant DAAL03-91-G-0035 from the Army Research Office. The research of the first author was also supported in part by a research grant from Cadre Technologies, Inc. The research of the second author was also supported in part by a National Science Foundation Presidential Young Investigator Award CCR–8906419, with matching funds from IBM Corporation, and by National Science Foundation research grant DCR–8403613.

time on an EREW PRAM and $O(\log n)$ time on a CREW PRAM with $M(n)$ processors [29], where the best known upper bound on $M(n)$ is currently $M(n) = O(n^{2.376})$ [10]. Transitive closure is a fundamental problem, and as a result much attention is given to reducing the required number of processors. The best previous results on the related problems of deciding the reachability of a vertex v from a vertex u (transitive closure query) and of computing a topological ordering of the vertices of an acyclic digraph G have the same time/processor bounds as transitive closure.

In the next section we discuss some important properties of planar st-graphs. In particular, we recall that a planar st-graph G admits two total orders on the set $V \cup E \cup F$, where V, E, and F are the sets of vertices, edges, and faces of G, respectively. Such total orders, denoted $<_{\text{L}}$ and $<_{\text{R}}$, provide an implicit representation of the transitive closure of G. Also, any such order yields a topological ordering of the vertices when restricted to V [51].

In § 3 we give an optimal $O(\log n)$-time, $(n/\log n)$-processor algorithm for constructing the orders $<_{\text{L}}$ and $<_{\text{R}}$ of an n-vertex planar st-graph G. This algorithm can be used as a preprocessing step to set up an $O(n)$-space data structure that supports transitive closure queries in $O(1)$ sequential time. Alternatively, we can construct within the same bounds a fully dynamic data structure that supports queries and updates (insertions/deletions of vertices and edges) in $O(\log n)$ sequential time. Using a different data structure, updates take $O(1)$ time with n processors and queries take $O(1)$ time with one processor. Since the publication of the conference version of this paper, Kao and Klein [28] have developed a transitive closure algorithm for general planar graphs that runs in $O(\log^3 n)$ time using n processors on a CRCW PRAM.

Section 4 considers the classical problem of point location in a planar subdivision, a fundamental searching primitive for a variety of geometric algorithms. We show how to preprocess a monotone subdivision in $O(\log n)$ time with $n/\log n$ processors on an EREW PRAM to obtain an $O(n)$-space data structure (the bridged separator tree [19], [33]) that supports point location queries in $O(\log n)$ time. Our technique can also be extended to construct a fully dynamic data structure for point location. Queries in the bridged separator tree can be done in optimal $O\big((\log n)/\log p\big)$ time using a p-processor CREW PRAM [56]. Nonmonotone subdivisions can be handled by our techniques by first applying a triangulation step, which takes $O(\log n)$ time using an n-processor CREW PRAM [3], [60].

Our results improve certain aspects of the previous best results [3], [9], [11], [12]. Atallah, Cole, and Goodrich [3] give an algorithm to construct a suboptimal $O(n \log n)$-space point location data structure in $O(\log n)$ time with n processors on a CREW PRAM. Dadoun and Kirkpatrick [11] show that the $O(n)$-space hierarchical point location data structure of Kirkpatrick [31] for triangulations can be constructed in $O(\log n \log^* n)$ worst-case time and $O(\log n)$ expected time on a CREW PRAM with n processors. A recent result of Cole and Zajicek [9] shows that the worst-case time can be reduced to $O(\log n)$ with $n/\log n$ processors at the expense of large constant factors. The hierarchical data structure can be modified so that it can process point location queries in $O\big((\log n)/\log p\big)$ time, but the required preprocessing takes $O(\log^2 n)$ time using $O(n^3)$ processors on a CREW PRAM [12]. An empirical analysis of the performance of several point-location data structures shows that the hierarchical point location data structure does not perform well in practice since the constant factors hidden behind the big-oh notation are large, whereas the bridged separator-tree constructed by our algorithm is very efficient [18].

In § 5, we investigate the problem of constructing visibility representations of

planar graphs, where the vertices are represented by horizontal segments and the edges by vertical segments. Such representations find applications in VLSI layout, motion planning, and graph drawing, and their combinatorial properties have been extensively investigated [16], [42], [52], [54], [58], [59]. We give algorithms for constructing visibility representations of planar *st*-graphs and undirected planar graphs in $O(\log n)$ time with $n/\log n$ processors. Also, we show that algorithms for drawing planar graphs that are based on the intermediate construction of visibility representations can be efficiently parallelized. We present algorithms that construct planar drawings with vertices placed at integer coordinates and asymptotically optimal area in $O(\log n)$ time with $n/\log n$ processors. This improves substantially over the algorithm of Ja'Ja' and Simon [26], which uses $M(n)$ processors to construct in $O(\log^2 n)$ time a planar drawing with vertices placed at real coordinates and no known bound on the area.

As a final remark, our parallel algorithms appear to be simple to implement and eminently practical.

2. Planar st-graphs.

DEFINITION 2.1. A *planar st-graph* G is a planar acyclic directed graph G with exactly one source vertex s and exactly one sink vertex t, which is embedded in the plane such that s and t are on the boundary of the external face.

An example is pictured in Fig. 1. We assume in this paper, as stated in Definition 2.1, that the input graph representation is embedded, that is, for each vertex the cyclical ordering of its neighbors is given. The embedding is represented in standard form by doubly-connected edge lists [38]. If the embedding information is not available, but a planar straight-line drawing is given, the embedding can be determined on an EREW PRAM in $O(\log d)$ time with n processors by sorting, where d is the maximum vertex degree [7]. This is optimal in the worst case, since sorting can be reduced to computing the embedding. If neither the embedding nor a drawing is given, the embedding can be determined as follows: We first add the directed edge (s,t) to G if it does not already exist. Let $\widehat{G}$ be the undirected planar graph corresponding to G. We can compute an embedding of $\widehat{G}$ on a CRCW PRAM in $O(\log n)$ time using the same number of processors needed to determine graph connectivity and to do bucket sorting in $O(\log n)$ time [40] ; the best known processor bound for this uses $n \log\log n$ processors deterministically [8], [24]. The resulting embedding is consistent with having any particular edge of $\widehat{G}$ appear on the external face, so we can assume that the edge (s,t), and thus vertices s and t, are on the external face. If the edge (s,t) was added in our construction earlier, it can be removed from $\widehat{G}$, and the orientations of the edges can be reintroduced to get an embedding of the planar *st*-graph G.

Following the development of Tamassia and Preparata [51], we will consider a planar embedding of G with s as the lowest vertex and t as the highest vertex, and with all edges directed upwards. Planar *st*-graphs have the following important properties [34], [52] :

1. Every vertex is on a directed path from s to t.
2. The incoming edges for each vertex appear consecutively around the vertex, and so do the outgoing edges. The face separating the incoming and outgoing edges of vertex v in the clockwise direction is called *left*(v), and the face separating them in the counterclockwise direction is called *right*(v). (See Fig. 2(a).)
3. The boundary of each face f consists of two directed paths enclosing f, each starting from the unique lowest vertex *low*(f) and ending at the unique high-

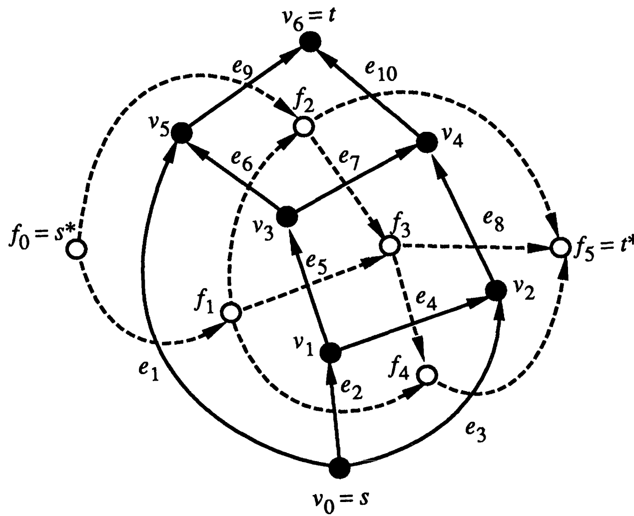


FIG. 1. *A planar st-graph G (solid lines) and its dual graph G^* (dashed lines).*

est vertex $high(f)$. (See Fig. 2(b).)

The terminology can be extended by defining vertices $low(x)$ and $high(x)$ and faces $left(x)$ and $right(x)$ for all elements in $V \cup E \cup F$, where V is the set of vertices, E is the set of edges, and F is the set of faces of G. For each vertex v, we define $low(v) = high(v) = v$ and $left(v)$ and $right(v)$ as above. For each edge $e = (u, v)$, we define $low(e) = u$, $high(e) = v$, and we define $left(e)$ to be the face to the left of e and $right(e)$ to be the face to the right of e. For each face f, we define $low(f)$ and $high(f)$ as above and $left(f) = right(f) = f$.

DEFINITION 2.2. The dual graph G^* of a planar st-graph G is the directed graph formed as follows: For each face of G, there is a vertex of G^*. In addition, the external face of G corresponds to two vertices s^* and t^* of G^*, which represent the "left" and "right" external faces of G. For each edge e in G, there is an edge $(left(e), right(e))$ in G^*. (See Fig. 1.)

It is easy to show that the dual graph G^* is also a planar st-graph. Partial orders $\uparrow$ and $\rightarrow$ can be defined on $V \cup E \cup F$ as follows.

DEFINITION 2.3. We say x is *below* y (denoted $x \uparrow y$) if there is a path from $high(x)$ to $low(y)$ in G, and we say x is *to the left of* y (denoted $x \rightarrow y$) if there is a path from $right(x)$ to $left(y)$ in the dual graph G^*.

For example, in Fig. 1, we have $e_2 \uparrow f_3 \uparrow t$ and $e_1 \rightarrow f_3 \rightarrow v_2$. For each $x, y \in V \cup E \cup F$, exactly one of the following relations holds: $x \uparrow y$, $y \uparrow x$, $x \rightarrow y$, or $y \rightarrow x$ [51]. This allows us to define the following two total orders.

DEFINITION 2.4. The total orders $<_{\text{L}}$ and $<_{\text{R}}$ are defined as

$$x <_{\text{L}} y \iff x \uparrow y \text{ or } x \rightarrow y;$$
$$x <_{\text{R}} y \iff x \uparrow y \text{ or } y \rightarrow x.$$

We define the *left sequence of G* to be the sequence of elements of $V \cup E \cup F$ sorted with respect to $<_{\text{L}}$, and the *right sequence of G* to be the sequence of elements of $V \cup E \cup F$ sorted with respect to $<_{\text{R}}$.

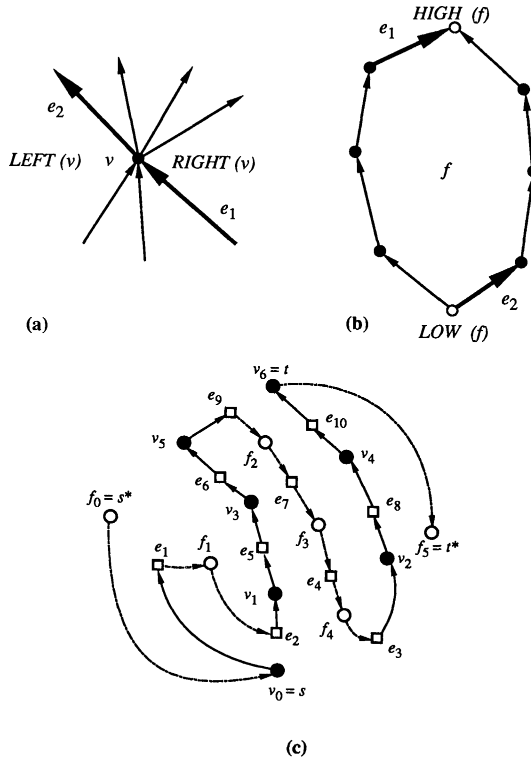


FIG. 2. *Parallel construction of the left sequence.* (a) *The order relations* $e_1 <_L v <_L e_2$ *formed by rule* 1. (b) *The order relations* $e_1 <_L f <_L e_2$ *formed by rule* 2. (c) *The left sequence in list form shown for the graph* G *in Fig.* 1.

For example, the left and right sequences for the graph in Fig. 1 are, respectively,

$$f_0\, v_0\, e_1\, f_1\, e_2\, v_1\, e_5\, v_3\, e_6\, v_5\, e_9\, f_2\, e_7\, f_3 e_4\, f_4\, e_3\, v_2\, e_8\, v_4\, e_{10}\, v_6\, f_5,$$

and

$$f_5\, v_0\, e_3\, f_4\, e_2\, v_1\, e_4\, v_2\, e_8\, f_3\, e_5\, v_3\, e_7\, v_4 e_{10}\, f_2\, e_6\, f_1\, e_1\, v_5\, e_9\, v_6\, f_0.$$

This left sequence is also pictured as a path in Fig. 2(c). The formal underpinning of the orders $<_L$ and $<_R$ can be found in the theory of planar lattices [27], [30].

The importance of the total orders $<_L$ and $<_R$ is that they can be used to answer transitive closure queries.

THEOREM 2.5 ([51]). *There is a path from vertex* u *to vertex* v *in a planar* st*-graph* G *if and only if* u *precedes* v *in both the left and right sequences of* G.

3. Transitive closure. The *transitive closure query problem* for a digraph G consists of answering queries of the form, "Is there a path from vertex u to vertex v in G?" In the dynamic problem, the digraph can be updated by insertions and deletions, and the queries can be interspersed with the updates. In this section, we exploit the properties of planar st-graphs and give EREW PRAM algorithms for constructing the fully dynamic (sequential) data structure of Tamassia and Preparata [51] in $O(\log n)$ time with $n/\log n$ processors. The data structure consists

of a pair of balanced trees associated with the left and right sequences and requires $O(n)$ space. When used sequentially, it is fully dynamic and handles queries and updates in $O(\log n)$ time. We also give parallel algorithms for dynamic queries and updates.

THEOREM 3.1. *Let G be a planar st-graph with n vertices. A fully dynamic data structure for the transitive closure query problem for G can be constructed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

Proof. Our algorithm constructs the data structure of Tamassia and Preparata [51] based on the left and right sequences of G. By Theorem 2.5, we can determine if there is a path from u to v in G by checking whether u is before v in both sequences. Each sequence is stored in the leaves of a balanced red-black tree [22]. Dynamic updates require a sequence of splits and splices in the tree.

Without loss of generality, let us restrict our attention to computing the left sequence of G. First we construct the dual graph G^*. The edges on the right boundary (respectively, left boundary) of each face f can identify a common representative vertex, say vertex $low(f)$, in parallel simultaneously for each face f, as follows: We construct a local order relation among the edges. If an edge is the leftmost (rightmost) edge incoming into a vertex, its successor is defined as the leftmost (rightmost) edge outgoing from that vertex. This order relation induces a set of ordered paths, corresponding to the right boundaries (left boundaries) of the faces. By list ranking [2, 8], the edges in the right boundary (left boundary) of each face can simultaneously identify a common vertex in $O(\log n)$ time with $O(n/\log n)$ processors.

To construct the left sequence of G, we note that, except for the very beginning and very end of the sequence, every other element in the sequence is an edge. We can form the sequence in $O(\log n)$ time with $O(n/\log n)$ processors by creating the following local order relations:

(1) Each vertex $v \neq s, t$ constructs the order relations $e_1 <_{\mathrm{L}} v <_{\mathrm{L}} e_2$, where e_1 is the rightmost incoming edge of v, and e_2 is the leftmost outgoing edge of v. (See Fig. 2(a).)

(2) Each interior face f constructs the two order relations $e_1 <_{\mathrm{L}} f <_{\mathrm{L}} e_2$, where e_1 is the topmost left edge of f, and e_2 is the bottommost right edge of f. (See Fig. 2(b).)

The source vertex s constructs the order relations $s^* <_{\mathrm{L}} s <_{\mathrm{L}} e_2$, where e_2 is the leftmost outgoing edge of s, and t forms the order relations $e_1 <_{\mathrm{L}} t <_{\mathrm{L}} t^*$, where e_1 is the rightmost incoming edge of t. List ranking is then done to combine the order relations into a fully ordered sequence, as shown in Fig. 2(c). Lemma 3.2 below shows that this sequence is the left sequence of G. The right sequence can be constructed analogously.

Given the left and right sequences of G, the dynamic data structure of Tamassia and Preparata [51] can be constructed easily in parallel. It consists of two balanced search trees, whose leaves consist of the elements of $V \cup E \cup F$. In one tree the leaves are ordered from left to right according to the left sequence, and in the other tree the leaves are ordered according to the right sequence. □

LEMMA 3.2. *List ranking of the above local order relations produces the left sequence of G.*

Proof. The local order relations produced above do not induce any cycles, since each order relation is consistent with the total order $<_{\mathrm{L}}$. The rest of the proof consists of showing by contradiction that the order relations induce a linear order on $V \cup E \cup F$. Suppose, during the application of the above two rules, that some edge e is chosen

twice as the head of two different subsequences. One of the subsequences must be formed by rule 1 above, and the other subsequence by rule 2, since two different vertices cannot have the same outgoing left edge, and two different faces cannot have the same bottommost right edge. Let us denote these two subsequences by $e' \, v \, e$ and $e'' \, f \, e$, for some vertex $v \neq s$ and some interior face f. By rule 1, e is the leftmost outgoing edge of v. By rule 2, e is the bottommost right edge of an interior face f, which implies that $low(f) = v$. This means that e is to the right of f, but there are no edges to the left of f, and hence f is not an interior face—a contradiction. We can show in a similar way that an edge cannot be chosen as the tail of two different subsequences. □

The fact that the total order $<_L$ is an extension of the partial order $\uparrow$ imposed by the directed edges of the graph gives us the following corollary.

COROLLARY 3.3. *A topological ordering of the n vertices of a planar st-graph G can be computed in $O(\log n)$ time using $n/\log n$ processors on an* EREW PRAM, *which is optimal. Specifically, we can compute the rank of each vertex in the vertex subsequence of the left or right sequence of G.*

Proof. First we compute the left-sequence (or right-sequence) of G, and then we extract the subsequence consisting of all the vertices by list ranking. □

Series-parallel graphs are a subclass of planar *st*-graphs, and thus we get the following corollary, which is an improvement over the $O(\log^2 n)$-time, n-processor CREW PRAM algorithm given by Afrati, Goldin, and Kanellakis [1].

COROLLARY 3.4. *Reachability in series-parallel graphs can be computed on an* EREW PRAM *in $O(\log n)$ time with $n/\log n$ processors.*

Our technique can be extended to solve the following problem posed by Kao in [28]: Given a planar *st*-graph G, compute for each vertex v the number of vertices reachable from v by paths in G. By associating each vertex v with a point $p(v)$ in the plane whose x- and y-coordinates are given by the ranks of v in the left and right sequences, respectively, we find that a vertex w is reachable from v if and only if the x- and y-coordinates of $p(w)$ are both greater than the corresponding ones of $p(v)$. Hence, we can apply the algorithm of Atallah, Cole, and Goodrich [3] for two-set dominance counting and obtain the following theorem.

THEOREM 3.5. *Given a planar st-graph G with n vertices, the number of vertices reachable from each vertex can be computed by an* EREW PRAM *in $O(\log n)$ time using n processors.*

The *contact chain query* problem for a convex subdivision and a direction θ consists of questions of the form: "If region r' is pushed in direction θ, will region r'' be moved?" [5]. Without loss of generality, assume that θ is the horizontal direction. By orienting the edges of the convex subdivision from bottom to top, and denoting by s and t the lowest and highest vertices, respectively, we get a planar *st*-graph G. (The subdivision is perturbed slightly if necessary to ensure that there are no horizontal edges.) It is easy to see that pushing r' will cause r'' to be moved if and only if there is a path from r' to r'' in the dual graph G^*. By Theorem 3.1 for the dual graph G^*, we get the following corollary.

COROLLARY 3.6. *A fully dynamic $O(n)$-space $O(\log n)$-sequential time data structure for the contact chain query problem along a fixed direction θ in an n-vertex convex subdivision can be constructed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

An alternate simple data structure for parallel use stores the left and right sequences as linear arrays. In array form, the shifts and swaps needed for dynamic

maintenance can clearly be done in constant time using n processors. If n^ϵ processors are available for updates, where $0 < \epsilon < 1$, the arrays can be replaced by B-trees [4] with nodes of degree $\Theta(n^\epsilon)$ and hence $O(1/\epsilon)$ height. This gives us the following result.

THEOREM 3.7. *Let G be a planar st-graph with n vertices. For any constant $0 < \epsilon \leq 1$, a data structure for the transitive closure query problem for G can be constructed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, such that a transitive closure query can be answered on an* EREW PRAM *in $O(1/\epsilon)$ time with one processor, and dynamic updates can be done in $O(1/\epsilon)$ time with n^ϵ processors.*

COROLLARY 3.8. *Let G be a planar st-graph with n vertices. After the preprocessing of Theorem* 3.7, *the subgraph H consisting of all paths from a vertex u to a vertex v can be generated in $O(\log n)$ time using $n/\log n$ processors on an* EREW PRAM, *and in constant time with n processors on a* CREW PRAM.

Proof. We assign one processor to each vertex and edge in the graph and broadcast the positions of u and v in $<_{\mathrm{L}}$ and $<_{\mathrm{R}}$ to all the processors. We form the desired subgraph H by including all the vertices and edges such that there is a path from u to v using that vertex or edge. This can be done using Theorem 2.5, by including all vertices and edges that come between u and v with respect to both $<_{\mathrm{L}}$ and $<_{\mathrm{R}}$. □

The shorter of the leftmost and rightmost paths from u to v can be generated in $O(\log n)$ time on an EREW PRAM, and in $O(\log k)$ time on a CREW PRAM, where k is the length of the path. We form the dual graph H^* of H. For each edge e in H, we test to see if it is on the leftmost (respectively, rightmost) path in H by checking if *left*(e) (respectively, *right*(e)) is not between u and v in either $<_{\mathrm{L}}$ or $<_{\mathrm{R}}$. This identifies the edges along the two paths. The shorter of the two paths can then be found by doing list ranking in parallel for each path.

4. Planar point location.

In this section, we present fast parallel algorithms for constructing data structures to handle point location queries. The queries themselves can be done either serially or in parallel using concurrent read. The reader is referred to the book of Preparata and Shamos [38] for the geometric terminology used in this section and a description of various point location techniques. Our approach is based on the *separator-method* for point location [19], [33].

DEFINITION 4.1. A *monotone chain* is a polygonal chain such that each horizontal line intersects it in at most one point. A polygon is *monotone* if its boundary is partitionable into two monotone chains. A (*planar*) *subdivision* S is a partition of the entire plane into polygons, called the *regions* of S. We assume a standard representation for the subdivision S and its embedding, such as a doubly-connected edge list representation [38]. A *monotone subdivision* is such that all its regions are monotone polygons.

A monotone subdivision S is therefore associated with a planar *st*-graph G, where each edge is directed according to increasing ordinate, and s and t are associated with the vertices at $-\infty$ and $+\infty$ of S. That is, an upward (respectively, downward) ray of S originating at vertex v corresponds to edge (v, t) (respectively, (s, v)) of G.

DEFINITION 4.2. Given a monotone subdivision S, a *separator* σ of S is a monotone chain of S between vertices at infinity, that is, a directed path of G from s to t. Given separators σ_1 and σ_2, we say that σ_1 is *to the left of* σ_2 if every horizontal line intersects σ_1 at or to the left of σ_2.

Let $r_1, r_2, \ldots, r_p$ be the regions of S, sorted according to some total order compatible with relation $\rightarrow$, that is, $r_i \rightarrow r_j$ implies $i < j$. The common boundary of the regions with index $\leq i$ and of the regions with index $> i$ is a separator of S,

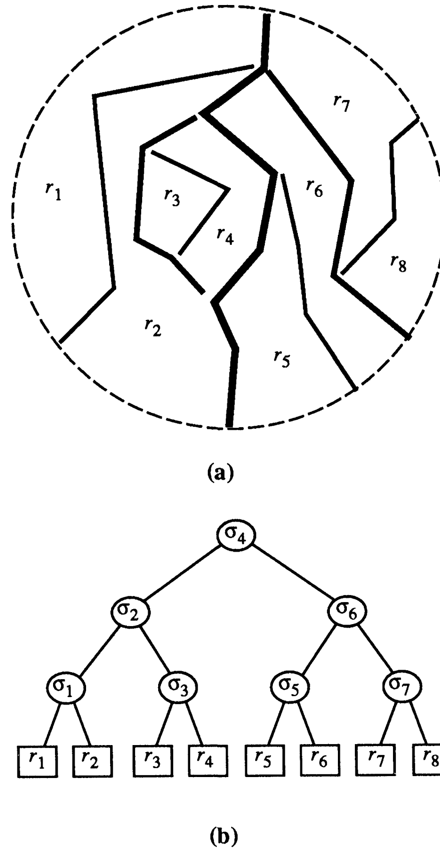


FIG. 3. *Construction of the separator tree for a regular subdivision.* (a) *Regular subdivision S with the chains of proper edges visualized.* (b) *Separator tree for S.*

which we denote σ_i. Clearly, σ_i is to the left of σ_j, for $i < j$.

One approach to point location is to perform a type of binary search on the set of separators $\Sigma = \{\sigma_1, \cdots, \sigma_{p-1}\}$, where each separator σ_i is assigned to a node (called *node σ_i*) of a balanced binary tree T (called the *separator tree*), whose leaves are the regions of S [33]. The sequence of the nodes of T in symmetric order is r_1, σ_1, r_2, σ_2, $\ldots$, σ_{p-1}, r_p. An edge (u, v) of S belongs to the interval of separators σ_i, σ_{i+1}, $\ldots$, σ_k such that $r_i = \mathit{left}(u, v)$ and $r_{k+1} = \mathit{right}(u, v)$; but for reasons of space efficiency (u, v) is stored only once, at node $\sigma_j = \mathit{lca}(r_i, r_{k+1})$, the lowest common ancestor of leaves r_i and r_{k+1}. The edges stored at a node σ_i, which are a subset of the edges of separator σ_i, are called the *proper edges* of σ_i. An example is shown in Fig. 3.

The separator tree uses $O(n)$ space and supports point location queries in $O(\log^2 n)$ time, where n is the number of vertices of S [33]. To perform a query, we trace a path in the separator tree from the root to the leaf r_i containing the query point q. At each internal node σ_i we discriminate q against separator σ_i and branch left or right according to whether q is to the left or right of σ_i. The discrimination of q against σ_i is performed by searching for the smallest value $\geq y(q)$ in the catalog of σ_i. The catalog consists of the y-coordinates of the proper edges of σ_i, along with

the dummy value $+\infty$. Each catalog entry is associated with the proper edge e (if it exists) whose top vertex has that y-coordinate. If the search for $y(q)$ returns the y-coordinate associated with edge e, then e is horizontally visible from q; we branch left if q is to the left of e, and right otherwise. When there is no edge associated with the y-coordinate returned, then $y(q)$ is in a "gap" between two proper edges of σ_i. In this case, the branching direction is determined as follows: If $y(q)$ is immediately above (respectively, below) proper edge e of σ_i, let σ_k be the ancestor of σ_i in the separator tree that stores the first nonproper edge of σ_i above (respectively, below) e. We branch left if σ_i is to the left of σ_k, and right otherwise. This information can be precomputed and stored in the catalog. Thus, the necessary branching can always be determined in constant time from the information associated with the y-coordinate returned as a result of the search in the catalog. Point location in this context consists merely of a sequence of catalog searches.

By applying the fractional cascading technique to the catalogs of the separator tree, we obtain a *bridged separator tree* (also called *layered dag*), which still uses $O(n)$ space and supports queries in $O(\log n)$ time, which is optimal [19]. (Our method in the previous paragraph for determining the branching in "gaps" yields a slight simplification of the algorithm.)

DEFINITION 4.3. A *regular* subdivision is a monotone subdivision having no pair of regions r and r' such that $r \uparrow r'$. (See Fig. 3.)

It follows that in a regular subdivision the relation $\rightarrow$ is a total order. Below, we show how to efficiently construct in parallel the bridged separator tree for a regular monotone subdivision, and then we extend the technique to arbitrary monotone subdivisions and general nonmonotone subdivisions.

LEMMA 4.4. *Each vertex of a regular subdivision has either indegree* 1 *or outdegree* 1.

Proof. If some vertex v of a regular subdivision has $indeg(v) \geq 2$ and $outdeg(v) \geq 2$, then there are regions r and r' such that $v = high(r) = low(r')$, which implies $r \uparrow r'$, a contradiction. □

The following algorithm constructs a bridged separator tree for a regular subdivision S. Without loss of generality, we assume that the number of regions p is a power of two.

(1) Construct the planar st-graph G associated with S, and compute its left and right sequences. Also, compute $indeg(v)$ and $outdeg(v)$ for each vertex v, and store with each edge (u, v) the indices i and j of the regions $r_i = left(u, v)$ and $r_j = right(u, v)$.

(2) Form a complete binary tree T whose leaves are associated with the regions of S (the faces of G), sorted from left to right according to their order in the left sequence of G. Hence, region r_i is the ith leaf from left to right. Also, construct an array of pointers to the internal nodes of T such that the ith element of the array points to the internal node of T associated with separator σ_i.

(3) Form the sets of proper edges of the internal nodes of T, as follows:

```
foreach edge (v, w) do begin
   if indeg(v) = 1
      then begin
         let (u, v) be v's only incoming edge;
         if lca(left(u, v), right(u, v)) = lca(left(v, w), right(v, w))
            then connect (u, v) to (v, w) bidirectionally
```

```
        end;
    if outdeg(w) = 1
        then begin
            let (w, z) be w's only outgoing edge;
            if lca(left(w, z), right(w, z)) = lca(left(v, w), right(v, w))
                then connect (w, z) to (v, w) bidirectionally
        end
    end;
```

(4) Store each doubly-connected list of edges obtained in step 3 into the node of T that is the lowest common ancestor of the regions to the left and right of all the edges in the list. Each list is the set of proper edges of that node, sorted from bottom to top.
(5) Convert the lists of proper edges into arrays, called *catalogs*, by means of list ranking. Establish bridges between the catalogs stored in adjacent nodes of T, according to the fractional cascading scheme of Atallah, Cole, and Goodrich [3].

The correctness of the algorithm follows from Lemma 4.4. Step 1 is performed using the techniques developed in the previous section. Step 2 can be easily done in $O(\log n)$ time with $n/\log n$ processors. In step 3, we use a simple technique for computing in $O(1)$ time the inorder rank of the lowest common ancestor of two leaves of a complete binary tree, given the ranks of such leaves in their left-to-right order [19]. Hence, the test

$$lca(left(u,v), right(u,v)) = lca(left(v,w), right(v,w))$$

can be done in $O(1)$ time using only the indices of the regions to the left and right of (u,v) and (v,w). By step 1, such indices are stored locally at the edges (u,v) and (v,w). Since the iterations of the **for**-loop are independent, we conclude that we can allocate one processor per group of $\log n$ edges and perform the computation of step 3 in $O(\log n)$ time with $n/\log n$ processors. In step 4, the assignment of the lists of proper edges to the corresponding internal nodes of T is done as follows. First, we pick any edge (u,v) of the list and compute in $O(1)$ time the rank of node $lca(left(u,v), right(u,v))$ in the symmetric order [19]. Next, from the rank, we access the node using the array constructed in step 2. Such computation can be performed in $O(\log n)$ time with $n/\log n$ processors.

The parallel fractional cascading technique of Atallah, Cole, and Goodrich [3] takes $O(\log n)$ time with $n/\log n$ processors to complete the construction of the bridged separator tree. This technique can be applied because, as described earlier, point location consists precisely of a series of catalog searches, where each node σ_i in the separator tree contains a catalog of y-coordinate values. One property of a regular subdivision is that the proper edges of each separator in the separator tree are connected, so that there are no "gaps" in the middle of a separator, but only at the top and bottom [39]. Thus, all but the first and last catalog entries are associated with a proper edge e of σ_i, and this simplifies the algorithm. This proves the following.

LEMMA 4.5. *Let S be a regular subdivision with n vertices. The bridged separator tree for point location in S can be constructed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

DEFINITION 4.6. We call two regions r' and r'' *vertically consecutive* if $r' \uparrow r''$ and there is no region r with $r' <_{\text{L}} r <_{\text{L}} r''$. It can be shown that there is a unique

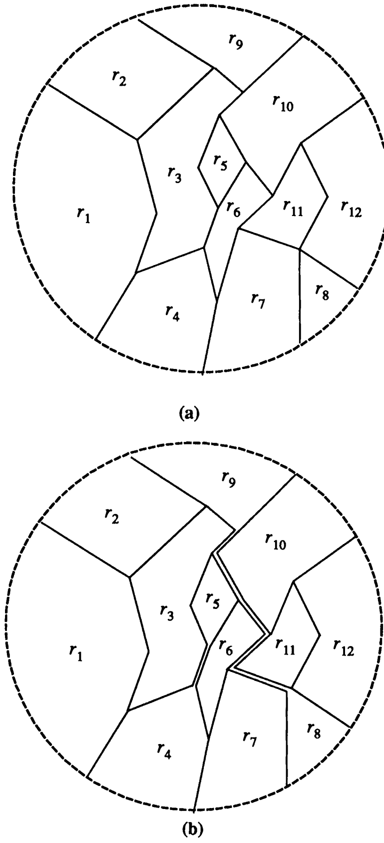


FIG. 4. (a) *A monotone subdivision* S *and* (b) *the corresponding regular subdivision* S^*. *Notice the clusters of regions* $r_4 \cup r_5$ *and* $r_8 \cup r_9$.

monotone chain from $high(r')$ to $low(r'')$, called a *channel*, and that all channels are vertex disjoint [39].

If the subdivision S is monotone, but not regular, we transform S into an equivalent regular subdivision by duplicating some edges [39]. Given two vertically consecutive regions r and r', we can imagine duplicating the channel from r to r', viewing the measure-zero region delimited by the two replicas as a degenerate polygon joining r and r' and merging them into a new region $r \cup r'$. By merging all sequences of vertically consecutive pairs in this way, we obtain a regular subdivision S^* whose regions are *clusters* of regions of S. (See Fig. 4.)

The algorithm for constructing subdivision S^* is as follows:

(1) Construct the planar *st*-graph G associated with S, and compute its left and right sequences.
(2) Extract the subsequence $r_1, r_2, \cdots, r_p$ of regions from the left sequence and determine the vertically consecutive pairs.
(3) For each vertically consecutive pair (r_i, r_{i+1}), mark the vertices and edges that are between $high(r_i)$ and $low(r_{i+1})$ in the left sequence.
(4) Duplicate all the vertices and edges that are marked and update the subdivision accordingly.

The transitive closure algorithm referred to in Theorem 3.1 is used for the preprocessing in step 1. The subsequence of regions can be formed using a standard binary tree communication scheme. We can verify whether two regions r_i and r_{i+1} are vertically consecutive by comparing the y-coordinates of vertices $high(r_i)$ and $low(r_{i+1})$. The remaining computations in the algorithm can be done easily in parallel. This proves the following.

LEMMA 4.7. *The regular subdivision S^* associated with a monotone subdivision S with n vertices can be computed by an* EREW PRAM *in $O(\log n)$ time with $n/\log n$ processors.*

The complete algorithm for preprocessing a monotone subdivision S consists of constructing S^* from S, and then building the bridged separator tree T^* for S^*. In practice, step 1 for constructing the bridged separator tree can be bypassed, since the ordered list of regions in S^*, sorted according to the left sequence, can be obtained directly from the corresponding list in S by contracting regions that are merged together into a cluster. The indegrees and outdegrees can be obtained directly also. Each leaf χ of T^* corresponds to a region of S^*, which in turn consists of some cluster of regions $r'_1, r'_2, \cdots, r'_k$ of S. We add to each leaf χ of T^* a pointer to a balanced search tree that stores the regions $r'_1, r'_2, \cdots, r'_k$, sorted from bottom to top.

To perform point location in S, we first determine the cluster χ containing the query point q by searching in T^*. Next, we search in the balanced tree pointed to by leaf χ in order to determine which region r_i of χ contains q. Hence, by combining the results of Lemmas 4.5 and 4.7, we obtain the following theorem.

THEOREM 4.8. *Let S be a monotone subdivision with n vertices. An $O(n)$-space data structure supporting $O(\log n)$-time point location queries in S can be constructed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

The algorithm used in Theorem 4.8 can be modified to construct the fully dynamic point location data structure of Preparata and Tamassia [39] within the same time/processor bounds.

For subdivisions that are represented without embedding information (e.g., by unsorted lists of vertices and edges), we need a preliminary step to compute its embedding, which consists of sorting the neighbors of each vertex v in clockwise order around v. This can be done in $O(\log n)$ time using n processors [7]. Note that if the embedding of S is not given as part of the input, there is an $\Omega(n \log n)$ lower bound on the amount of work needed to compute the embedding in the worst case [32].

For nonmonotone subdivisions we perform a preliminary triangulation step and then apply the technique for monotone subdivisions. Triangulation can be performed by a CREW PRAM in $O(\log n)$ time with n processors [3], [60].

We get the following theorem.

THEOREM 4.9. *Let S be a subdivision with n vertices. An $O(n)$-space data structure supporting $O(\log n)$-time point location queries in S can be constructed by a* CREW PRAM *in $O(\log n)$ time using n processors.*

The bridged separator tree data structure can also be used to process the queries in parallel. We show in a companion paper [56] that an $O(n)$-space data structure can be constructed with an EREW PRAM in $O(\log n)$ time using $n/\log n$ processors such that, for any $2 \leq p \leq n$, point location queries can be done in $O\big((\log n)/\log p\big)$ time using a CREW PRAM with p processors. This algorithm improves upon the one of Dadoun and Kirkpatrick [12]. It achieves the same query time, but it is simpler and uses less preprocessing. The query time of $O\big((\log n)/\log p\big)$ is optimal since we can reduce the problem of dictionary searching to planar point location, and thus the

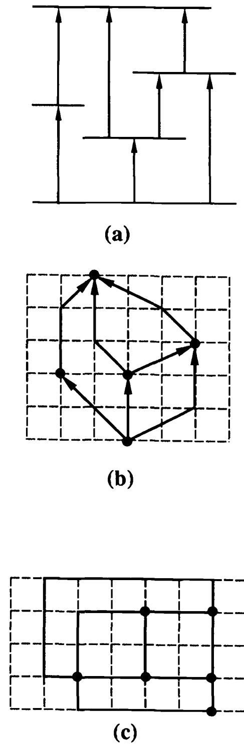


FIG. 5. (a) *Visibility representation for a planar st-graph G.* (b) *A planar upward polyline grid drawing of G.* (c) *A planar orthogonal grid drawing of an undirected graph.*

lower bound of Snir [46] applies.

5. Visibility representations and graph drawing. The concept of *visibility* plays a fundamental role in a variety of geometric problems and applications, such as art gallery problems [35]; VLSI layout [25], [44], [57]; motion planning [23], [41]; and graph drawing [13], [53].

DEFINITION 5.1. Given a collection H of horizontal segments in the plane, the (vertical) *visibility graph* of H is the graph G whose vertices are the segments of H and whose edges are pairs of segments that see each other in the vertical direction. The edges of G can be oriented from bottom to top to yield an acyclic digraph.

DEFINITION 5.2. A *visibility representation* Γ for a directed graph G maps each vertex v of G to a horizontal segment $\Gamma(v)$ and each edge (u, v) to a vertical segment $\Gamma(u, v)$ that has its lower endpoint on $\Gamma(u)$, its upper endpoint on $\Gamma(v)$, and does not intersect any other horizontal segment. (See Fig. 5(a).) If G is an undirected planar graph, a visibility representation for G is defined as a visibility representation for some orientation of G.

Besides having many applications, visibility graphs and representations are also of intrinsic theoretical interest, and their combinatorial properties have been extensively investigated [16], [52], [54], [58], [59].

The visibility graph of a set of n segments can be computed in $O(n \log n)$ sequential time and $O(n)$ space [44], which is optimal. It can also be constructed in parallel by an EREW PRAM in $O(\log n)$ time and $O(n \log n)$ space with n processors [3], or

in $O(\log^2 n)$ time and $O(n)$ space with $n/\log n$ processors [43]. As regards visibility representations, there are sequential $O(n)$-time algorithms for their construction [13], [42], [52].

THEOREM 5.3. *Let G be a planar st-graph with n vertices. A visibility representation for G with integer coordinates and $O(n^2)$ area can be computed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

Proof. A visibility representation for G can be constructed by the following variation of previous algorithms [13], [42], [52].

(1) Compute a topological ordering $Y(v)$ of the vertices of G.
(2) Compute a topological ordering $X(f)$ of the vertices of G^*, the dual graph of G.
(3) Draw each vertex-segment $\Gamma(v)$ at ordinate $Y(v)$ and between abscissae $X(\mathit{left}(v))$ and $X(\mathit{right}(v)) - 1$.
(4) Draw each edge-segment $\Gamma(e)$ at abscissa $X(\mathit{left}(e))$ and between ordinates $Y(\mathit{low}(e))$ and $Y(\mathit{high}(e))$.

By Corollary 3.3, steps 1 and 2 take $O(\log n)$ time using $n/\log n$ processors. The parallel computation of steps 3 and 4 within the same bounds is straightforward. □

Given a 2-connected embedded undirected planar graph G, we choose s and t to be two adjacent vertices (which we can assume to be on the external face) and orient the edges of G so that the resulting digraph is a planar st-graph, and then we apply the previous theorem. Such an orientation of G can be computed by an EREW PRAM in $O(\log n)$ time with $n/\log n$ processors using the st-numbering algorithm of Gazit [21].

THEOREM 5.4. *Let G be a 2-connected embedded (undirected) planar graph with n vertices. A visibility representation for G with integer coordinates and $O(n^2)$ area can be computed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors.*

A number of data presentation problems involve drawing graphs so that they are easy to read and understand. Examples include circuit schematics, algorithm animation, and diagrams for information systems analysis and design. The literature on graph drawing algorithms is spread over the broad spectrum of computer science [17], [50]. This problem has received increasing theoretical interest in recent years (cf. [15], [20], [45]).

DEFINITION 5.5. A *drawing* of a graph maps each vertex into a point of the plane, and each edge (u, v) into a simple open curve between the points associated with the vertices u and v. A *planar drawing* has no crossing edges. A *straight-line drawing* is such that every edge is drawn as a line segment. In a *polyline drawing*, every edge is drawn as a polygonal chain. An *orthogonal drawing* is a polyline drawing whose edges are chains of horizontal and vertical segments. A *grid drawing* is a polyline drawing such that the vertices and the bends of the edges have integer coordinates. An *upward drawing* for an acyclic digraph G is such that every edge (u, v) is a curve monotonically increasing in the vertical direction. (See examples in Figs. 5(b), (c).)

An edge (u, v) of a digraph is said to be *transitive* if there exists a directed path from u to v that does not contain the edge (u, v). A digraph is said to be *reduced* if it has no transitive edges. A reduced planar st-graph G admits a planar upward straight-line drawing such that the x- and y-coordinates of a vertex v are the ranks of v in the restriction to the vertices of the left- and right-sequence of G, respectively [15]. Hence, a reduced planar st-graph can be efficiently drawn in parallel from the result of Corollary 3.3.

To draw a nonreduced planar *st*-graph we insert a new dummy vertex v along each transitive edge (u, w) and draw the resulting reduced planar *st*-graph G' considering the dummy vertices as bends. To identify transitive edges in parallel we use the following lemma, where we say that edge (u, v) is the *long edge* of face f if $u = low(f)$ and $v = high(f)$.

LEMMA 5.6. *An edge e of a planar st-graph is transitive if and only if it is the long edge of either left(e) or right(e).*

By Euler's formula a planar graph has at most $2n - 5$ interior faces, so that Lemma 5.6 implies that a planar *st*-graph has at most $2n - 5$ transitive edges.

Hence, we have the following theorem.

THEOREM 5.7. *Let G be a planar st-graph with n vertices. A planar upward polyline grid drawing for G with $2n - 5$ bends and $O(n^2)$ area can be computed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors, which is optimal.*

Now, we consider planar orthogonal drawings of undirected graphs. Such drawings are typical of circuit layout, and are widely used in data presentation applications because of their regularity. Sequential algorithms for planar orthogonal drawings are given by Storer [47], Tamassia [48], and Tamassia and Tollis [53].

THEOREM 5.8. *Let G be a 2-connected embedded (undirected) planar graph with n vertices, each of degree at most four. A planar orthogonal grid drawing for G with $O(n)$ bends and $O(n^2)$ area can be computed by an* EREW PRAM *in $O(\log n)$ time using $n/\log n$ processors.*

Proof. As shown by Tamassia and Tollis [53], a planar orthogonal grid drawing can be constructed from a visibility representation by local replacements performed at each vertex. Because of its locality, this transformation can be easily parallelized. Hence, the result follows from Theorem 5.4. □

The bounds on the area and the number of bends are asymptotically optimal [47]. The bound on the number of bends can be improved to the exact worst-case optimal $2n + 4$ and the algorithm can be extended to 1-connected graphs [55].

Our results improve upon the previous parallel drawing algorithm presented by Ja'Ja' and Simon [26], which constructs a straight-line planar drawing in $O(\log^2 n)$ time with $M(n)$ processors, using real arithmetic for the computation of the coordinates of the vertices. It is not known whether this algorithm can be modified to construct grid drawings with area bounded by a polynomial in n.

Acknowledgments. We would like to thank the referees for several useful comments and suggestions.

REFERENCES

[1] F. N. AFRATI, D. Q. GOLDIN, AND P. C. KANELLAKIS, *Efficient parallelism for structured data: Directed reachability in* S-P DAGS, Tech. Report CS-88-07, Department of Computer Science, Brown University, Providence, RI, 1988.

[2] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in VLSI Algorithms and Architectures, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 81–90.

[3] M. J. ATALLAH, R. COLE, AND M. T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM J. Comput., 18 (1989), pp. 499–532.

[4] R. BAYER AND E. M. MCCREIGHT, *Organization and maintenance of large ordered indices*, Acta Informatica, 1 (1972), pp. 173–189.

[5] B. CHAZELLE, H. EDELSBRUNNER, AND L. J. GUIBAS, *The complexity of cutting convex polytopes*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 66–76.

[6] N. CHIBA, T. NISHIZEKI, S. ABE, AND T. OZAWA *A linear algorithm for embedding planar graphs using* PQ-*trees*, J. Comput. System Sci., 30 (1985), pp. 54–76.

[7] R. Cole, *Parallel merge sort*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511–516.

[8] R. Cole and U. Vishkin, *Approximate and exact parallel scheduling with applications to list, tree, and graph problems*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 478–491.

[9] R. Cole and O. Zajicek, *An optimal parallel algorithm for building a data structure for point location*, J. Parallel Distributed Comput., to appear.

[10] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progression*, in Proc. 28th ACM Symposium on Theory of Computing, 1987, pp. 1–6.

[11] N. Dadoun and D. G. Kirkpatrick, *Parallel processing for efficient subdivision search*, in Proc. 3rd ACM Symposium on Computational Geometry, 1987, pp. 205–214.

[12] ———, *Cooperative subdivision search algorithms with applications*, in Proc. 27th Annual Allerton Conference, Monticello, IL, 1989, pp. 538–547.

[13] G. Di Battista and R. Tamassia, *Algorithms for plane representations of acyclic digraphs*, Theoret. Comput. Sci., 61 (1988), pp. 175–198.

[14] ———, *Incremental planarity testing*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 436–441.

[15] G. Di Battista, R. Tamassia, and I. G. Tollis, *Area requirement and symmetry display in drawing graphs*, in Proc. 5th ACM Symposium on Computational Geometry, 1989, pp. 51–60.

[16] P. Duchet, Y. Hamidoune, M. Las Vergnas, and H. Meyniel, *Representing a planar graph by vertical lines joining different levels* Discrete Math., 46 (1983), pp. 319–321.

[17] P. Eades and R. Tamassia, *Algorithms for automatic graph drawing: An annotated bibliography*, Tech. Report CS-89-09, Department of Computer Science, Brown University, Providence, RI, 1989.

[18] M. Edahiro, I. Kokubo, and T. Asano, *A new point-location algorithm and its practical efficiency—comparison with existing algorithms*, ACM Trans. Graphics, 3 (1984) pp. 86–109.

[19] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[20] H. de Fraysseix, J. Pach, and R. Pollack, *Small sets supporting Fary embeddings of planar graphs*, in Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 426–433.

[21] H. Gazit, *Optimal* EREW *parallel algorithms for connectivity, ear decomposition, and st-numbering of planar graphs*, manuscript, Department of Computer Science, Duke University, Durham, NC, 1990

[22] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, in Proc. 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

[23] L. J. Guibas and F. F. Yao, *On translating a set of rectangles*, in Advances in Computing Research, Vol. 1, F. P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1983, pp. 61–77.

[24] T. Hagerup, *Towards optimal parallel bucket sorting*, Tech. Report 02/1987, Universität des Saarlandes, Saarbrücken, Germany, January 1987.

[25] M. Y. Hsueh and D. O. Pederson, *Computer-aided layout of* LSI *circuit building-blocks*, in Proc. IEEE Internat. Symposium on Circuits and Systems, 1979, pp. 474–477.

[26] J. Ja'Ja' and J. Simon, *Parallel algorithms in graph theory: Planarity testing*, SIAM J. Comput., 11 (1982), pp. 314–328.

[27] T. Kameda, *On the vector representation of the reachability in planar directed graphs*, Inform. Process. Lett., 3 (1975), pp. 75–77.

[28] M.-Y. Kao and P. N. Klein, *Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs*, in Proc. 22nd ACM Symposium on Theory of Computing, 1990, pp. 181–192.

[29] R. M. Karp and V. Ramachandran, *A survey of parallel algorithms for shared memory machines*, in Handbook of Theoretical Computer Science, North-Holland, Amsterdam, 1990.

[30] D. Kelly and I. Rival, *Planar lattices*, Canad. J. Math., 27 (1975), pp. 636–665.

[31] D. G. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[32] ———, *Establishing order in planar subdivisions*, Discrete & Comput. Geom., 3 (1988), pp. 267–280.

[33] D. T. Lee and F. P. Preparata, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[34] A. Lempel, S. Even, and I. Cederbaum, *An algorithm for planarity testing of graphs*, in Theory of Graphs, Internat. Symposium, Rome, Italy, 1966, pp. 215–232.

[35] J. O'ROURKE, *Art Gallery Theorems and Algorithms*, Oxford University Press, London, 1987.
[36] R. H. J. M. OTTEN AND J. G. VAN WIJK, *Graph representations in interactive layout design*, in Proc. IEEE Internat. Symposium on Circuits and Systems, 1978, pp. 914–918.
[37] T. OZAWA AND H. TAKAHASHI, *A graph-planarization algorithm and its applications to random graphs*, in Graph Theory and Algorithms, Lecture Notes in Computer Science 108, Springer-Verlag, Berlin, New York, 1981, pp. 95–107.
[38] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, Berlin, New York, 1985
[39] F. P. PREPARATA AND R. TAMASSIA, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.
[40] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, in Proc. IEEE Symposium on Foundations of Computer Science, 1989, pp. 282–287
[41] I. RIVAL AND J. URRUTIA, *Representing orders by translating convex figures in the plane*, Order, 4 (1988), pp. 319–339.
[42] P. ROSENSTIEHL AND R. E. TARJAN, *Rectilinear planar layouts of planar graphs and bipolar orientations*, Discrete & Comput. Geom., 1 (1986), pp. 343–353.
[43] J. E. SAVAGE AND M. G. WLOKA, *Parallel constraint graph generation*, in Proc. Decennial Caltech Conference on VLSI, MIT Press, Cambridge, MA, 1989, pp. 241–259.
[44] M. SCHLAG, F. LUCCIO, P. MAESTRINI, D. T. LEE, AND C. K. WONG, *A visibility problem in* VLSI *layout compaction*, in Advances in Computing Research, Vol. 2, F. P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1985, pp. 259–282.
[45] W. SCHNYDER, *Embedding planar graphs on the grid*, in Proc. 1st ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 138–148.
[46] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1989), pp. 688–708.
[47] J. A. STORER, *On minimal node-cost planar embeddings*, Networks, 14 (1984), pp. 181–212.
[48] R. TAMASSIA, *On embedding a graph in the grid with the minimum number of bends*, SIAM J. Comput., 16 (1987), pp. 421–444.
[49] ———, *A dynamic data structure for planar graph embedding*, in Automata, Languages and Programming, Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, New York, 1988, pp. 576–590.
[50] R. TAMASSIA, G. D. BATTISTA, AND C. BATINI, *Automatic graph drawing and readability of diagrams*, IEEE Trans. Systems Man Cybernet., 18 (1988), pp. 61–79.
[51] R. TAMASSIA AND F. P. PREPARATA, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.
[52] R. TAMASSIA AND I. G. TOLLIS, *A unified approach to visibility representations of planar graphs* Discrete & Comput. Geom., 1 (1986), pp. 321–341.
[53] ———, *Planar grid embedding in linear time*, IEEE Trans. Circuits and Systems, 36 (1989), pp. 1230–1234.
[54] ———, *Representations of graphs on a cylinder*, SIAM J. Discrete Math., 4 (1991), pp. 139–149.
[55] R. TAMASSIA, I. G. TOLLIS, AND J. S. VITTER, *Parallel construction of planar graph layouts*, manuscript, 1990.
[56] R. TAMASSIA AND J. S. VITTER, *Optimal cooperative search in fractional cascaded data structures*, Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures, 1990 pp. 307–316.
[57] S. WIMER, I. KOREN, AND I. CEDERBAUM, *Floorplans, planar graphs, and layouts*, IEEE Trans. Circuits and Systems, 35 (1988), pp. 267–278.
[58] S. K. WISMATH, *Characterizing bar line-of-sight graphs*, Proc. 1st ACM Symposium on Computational Geometry, 1985, pp. 147–152.
[59] ———, *Weighted visibility graphs of bars and related flow problems*, in Algorithms and Data Structures, Lecture Notes in Computer Science 382, Springer-Verlag, Berlin, New York, 1989, pp. 325–334.
[60] C. K. YAP, *Parallel triangulation of a polygon in two calls to the trapezoidal map*, Algorithmica, 3 (1988), pp. 279–288.

SIAM J. COMPUT.
Vol. 20, No. 4 , pp. 726–736, August 1991

© 1991 Society for Industrial and Applied Mathematics
009

ON NONBLOCKING MULTIRATE INTERCONNECTION NETWORKS*

SHUN-PING CHUNG† AND KEITH W. ROSS‡

Abstract. In a recent paper, Melen and Turner [*SIAM J. Comput.*, 18 (1989), pp. 301–313] determined sufficient conditions for multirate interconnection networks to be strictly nonblocking and rearrangeable. They considered the continuous bandwidth case, which permits the bandwidth of a connection to take an arbitrary value in a given closed interval. In this paper, simple necessary and sufficient conditions for multirate interconnection networks to be strictly nonblocking for both discrete and continuous bandwidth cases are determined. New results for rearrangeable multirate networks with discrete bandwidth requirements are also given.

Key words. interconnection networks, telecommunications, Clos network, Beneš network, Cantor network

AMS(MOS) subject classifications. 94A99, 68E10, 05B35

1. Introduction. In a recent paper, Melen and Turner [1] introduced an elegant model for interconnection networks that carry multirate traffic. The impetus of such a model comes from the current interest in designing telecommunication switches that handle traffic with a wide range of bandwidth requirements (voice, facsimile, video, etc.) Their *continuous bandwidth case* permits the bandwidth of a connection to take an arbitrary value in a given closed interval. Their *single path model* requires that the entire bandwidth of a connection be routed through a single path in the interconnection network. Melen and Turner give *sufficient* conditions for their model to be strictly nonblocking and rearrangeable. For example, generalizing the well-known result of Clos [5], they determine the number of middle-stage switches in a three-stage multirate network that would ensure strictly nonblocking operation. In another recent paper, Niestegge [9] determines sufficient conditions for a Clos network to be strictly nonblocking for the *discrete bandwidth case*, i.e., for the bandwidth of all connections to belong to a given finite set.

In this paper we determine simple *necessary and sufficient* conditions for multirate interconnection networks to be strictly nonblocking. We do this for the continuous bandwidth case as well as for a discrete bandwidth case. We also consider rearrangeable multirate networks for the discrete case.

In §3 we consider three-stage Clos networks. The minimum number of middle-stage switches for strictly nonblocking operation is determined for the discrete case. The continuous case is studied under the condition that the maximum bandwidth requirement of a connection is equal to the capacity of the edges in the interconnection network. Again, the minimum number of middle-stage switches is obtained.

In §4 we consider Cantor networks. Employing polymatroid theory, we determine the minimum number of subnetworks required for nonblocking operation for both discrete and continuous cases. We show for both Clos and Cantor networks that the sufficient conditions of [1] can be loose in many circumstances.

In §5 we consider rearrangeable networks. We establish that the Beneš network

* Received by the editors August 1, 1989; accepted for publication (in revised form) October 7, 1990. This research was partially supported by both AT&T grant 5-27628 and National Science Foundation grant NCR-8707620.

† Department of Electrical Engineering, University of Pennsylvania, Philadelphia, Pennsylvania 19104.

‡ Department of Systems, University of Pennsylvania, Philadelphia, Pennsylvania 19104.

remains rearrangeable if the bandwidth requirements of the connections are the same, but less than the capacity of the edges in the network. We also show by way of counterexamples that the Beneš network is not generally rearrangeable for multirate traffic. More complex networks are then constructed that are rearrangeable for multirate traffic.

2. Model description. For the sake of presentation, we recapitulate the multirate interconnection network model of Melen and Turner. We then formally define the continuous and discrete bandwidth cases.

The interconnection network shall be viewed as a directed graph $G = (V, E)$ which includes a set of input nodes I and output nodes O, where each input node has one outgoing edge and no incoming edge, and each output node has one incoming edge and no outgoing edge. Only networks that can be divided into a sequence of stages shall be considered. Input ports are in stage 0 and for $i > 0$, a node v is in stage i if for all links (u, v), u is in stage $i-1$. A link (u, v) is said to be in stage i if u is in stage i. In the networks considered here, all output ports are in the same stage, and no other nodes are in this stage. When referring to a k-stage network, we generally neglect the stages containing the input and output ports.

A connection in a network is a triple (x, y, ω), where $x \in I$, $y \in O$, and $0 \leq \omega \leq 1$. The weight ω represents the bandwidth required by the connection. A route is a path joining an input node to an output node, with intermediate nodes in $V - (I \cup O)$, together with a weight. A route r realizes a connection (x, y, ω), if x and y are the input and output nodes joined by r and the weight of r equals ω.

A set of connections is said to be compatible if for all nodes $x \in I \cup O$, the sum of the weights of all connections involving x is ≤ 1. A configuration for a network G is a set of routes. The weight on an edge in a particular configuration is the sum of the weights of all the routes passing through that edge. A configuration is compatible if for all edges $(u, v) \in E$, the weight on (u, v) is ≤ 1. A set of connections is said to be realizable if there is a compatible configuration that realizes that set of connections. If we are attempting to add a connection (x, y, ω) to an existing configuration, we say that a node u is accessible from x if there is a path from x to u, all of whose edges have a weight of no more than $1 - \omega$.

A network is said to be rearrangeable if for every set C of compatible connections, there exists a compatible configuration that realizes C. A network is strictly nonblocking if for compatible configuration R, realizing a set of connections C, and every connection c compatible with C, there exists a route r that realizes c and is compatible with R. We will consider two cases :

- *Discrete bandwidth case.* The weight of all connections belongs to a given finite set $\{b_1, \cdots, b_K\}$, where b_1 is a divisor of b_k, $k = 2, \cdots, K$. Denote $b := b_1$ and $B := \max\{b_k : k = 1, \cdots, K\}$.
- *Continuous bandwidth case.* The weight of all connections belongs to a closed interval $[b, B]$, where $0 \leq b \leq B \leq 1$.

In order to simplify notation, we shall always suppose that $1/b$ is an integer for the discrete bandwidth case. (If $1/b$ is not an integer, then the analysis to follow for the discrete bandwidth case can be modified with little effort.) However, we will not impose this restriction for the continuous bandwidth case.

Suppose that b is a divisor of B. If a given interconnection network is strictly nonblocking for the continuous bandwidth case, then it will be strictly nonblocking for any discrete bandwidth case with $b_1 = b$ and $B = \max\{b_k : k = 1, \cdots, K\}$. Note that the classical circuit switching case corresponds to $b = 1$ for the discrete bandwidth

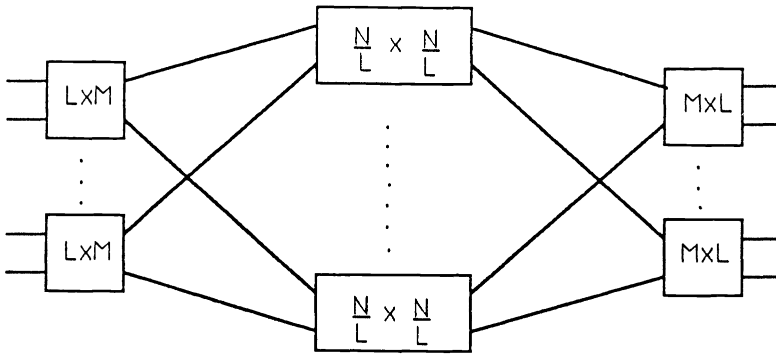


FIG. 1. *Clos network* $C_{N,L,M}$.

case and $b > \frac{1}{2}$ for the continuous bandwidth case.

3. Three-stage Clos networks. A three-stage Clos network with N input ports and N output ports is depicted in Fig. 1. Note that there are L input ports (respectively, output ports) for each first-stage node (respectively, last-stage node). Also note that there are M middle-stage nodes. The three-stage Clos network shall be denoted by $C_{N,L,M}$. Let M^* be the minimum number of middle-stage nodes for $C_{N,L,M}$ to be strictly nonblocking (with N and L held fixed). It is well known [3] that $M^* = 2L - 1$ for classical circuit switching. In §§3.1–3.3 we determine M^* for the multirate model for both the discrete and continuous bandwidth cases.

3.1. The discrete bandwidth case.

THEOREM 1. *For the discrete bandwidth case,*

$$M^* = 2\left\lfloor \frac{L-B}{1-B+b} \right\rfloor + 1. \tag{1}$$

Proof. Let M^* be given by (1). Suppose we want to add a connection (x, y, ω) to an arbitrary configuration. The divisibility condition implies that the minimun weight on an edge needed to block this connection is $s(\omega) = 1 - \omega + b$. The argument in the proof of Proposition 3.1 of [1] and the argument in [9] show that the maximun number of inaccessible middle-stage nodes from either x or y is

$$2\left\lfloor \frac{L-\omega}{s(\omega)} \right\rfloor,$$

which is maximized at $\omega = B$. Therefore, since M^* is given by (1), there must be at least one middle-stage node that is accessible from both x and y, implying that C_{N,L,M^*} is strictly nonblocking.

It remains to show that if $M = M^* - 1$, then the Clos network $C_{N,L,M}$ is not strictly nonblocking. Consider the following configuration with $2(L - B)/b$ connections, each with weight b. $(L - B)/b$ of these connections employ the same first-stage node u and the same last-stage node z, but they contribute a weight of at least $1-B+b$ to each of $\lfloor (L - B)/(1 - B + b) \rfloor$ middle-stage nodes. The remaining $(L - B)/b$ connections employ the same first-stage node $w \neq u$ and the same last-stage node $v \neq z$, but they contribute a weight of at least $1 - B + b$ to $\lfloor (L - B)/(1 - B + b) \rfloor$ middle-stage nodes. Define the configuration so that the two sets of middle-stage nodes are

disjoint (this is possible since $M = 2\lfloor (L-B)/(1-B+b) \rfloor$). Then for a compatible connection (x, y, B) with x adjacent to u and y adjacent to v, a middle-stage node would not be available and the connection would be blocked. Thus, $C_{N,L,M}$ is not strictly nonblocking. □

3.2. The continuous bandwidth case.

THEOREM 2. *For the continuous bandwidth case with $B = 1$,*

$$M^* = 2\lfloor 1/b \rfloor (L-1) + 1. \tag{2}$$

In order to prove Theorem 2, consider a network with *one* node (in $V - (I \bigcup O)$) with L input ports and $\lfloor 1/b \rfloor L$ output ports. Suppose that the connections for this network have arbitrary weights in $[b, 1]$ (i.e., the continuous bandwidth case). Note that this network is strictly nonblocking. Fix $\omega \in [b, 1]$, and let $\mathcal{R}$ be the set of all configurations such that the first input edge has a weight $\leq 1 - \omega$. Let $J(\omega, L)$ be the maximum number of output edges that has a weight $> 1 - \omega$, where the maximization is over all configurations in $\mathcal{R}$. The proof of Theorem 2 hinges on the following technical result.

LEMMA 1. (i) $J(\omega, L) \leq \lfloor 1/b \rfloor (L-1)$;

(ii) $J(1, L) = \lfloor 1/b \rfloor (L-1)$.

Proof. (i) Let R be a configuration in $\mathcal{R}$. For each route $r \in R$, denote α_r for its weight. Let G_l be the set of routes in R that pass through the lth input edge. Thus, $\{G_1, \cdots, G_L\}$ is a partition of R. Let $\mathcal{J}$ be the set of all output edges that have weight $> 1 - \omega$ and let $J = |\mathcal{J}|$. Let H_j be the set of routes in R that pass through the jth edge in $\mathcal{J}$. Then α_r, $r \in R$, must satisfy

$$\sum_{r \in G_1} \alpha_r \leq 1 - \omega, \tag{3}$$

$$\sum_{r \in G_l} \alpha_r \leq 1, \qquad l = 2, \cdots, L, \tag{4}$$

$$\sum_{r \in H_j} \alpha_r > 1 - \omega, \qquad j \in \mathcal{J}, \tag{5}$$

$$\alpha_r \in [b, 1], \qquad r \in R. \tag{6}$$

Let $G := \bigcup_{l=2}^{L} G_l$, we have

$$|H_j \cap G| \geq 1, \qquad j \in \mathcal{J}. \tag{7}$$

Otherwise, for some $j \in \mathcal{J}$, $H_j \subseteq G_1$, and thus from (3),

$$\sum_{r \in H_j} \alpha_r \leq \sum_{r \in G_1} \alpha_r \leq 1 - \omega, \tag{8}$$

contradicting (5). From (7) we have

$$J \leq \sum_{j \in \mathcal{J}} |H_j \cap G| \leq |G|. \tag{9}$$

From (4) and (6) we have $|G_l| \leq \lfloor 1/b \rfloor$, $l = 2, \cdots, L$ so that

$$|G| \leq \lfloor 1/b \rfloor (L-1). \tag{10}$$

Combining (9) and (10) gives the desired result.

(ii) Consider a configuration R consisting of $\lfloor 1/b \rfloor (L-1)$ routes of weight b, where each route passes through a different output edge. Further define R so that $\lfloor 1/b \rfloor$ routes pass through the lth input edge, $l = 2, \cdots, L$. If $\omega = 1$, then each of the $\lfloor 1/b \rfloor (L-1)$ edges utilized by R have a weight $> 1 - \omega$. Hence, $J(1, L) \geq \lfloor 1/b \rfloor (L-1)$. □

Proof of Theorem 2. We first show that C_{N,L,M^*} is strictly nonblocking with M^* given by (2). Suppose we want to add a connection (x, y, ω) to an arbitrary configuration R. It follows from Lemma 2 that at most $\lfloor 1/b \rfloor (L-1)$ middle-stage nodes are inaccessible from x and at most $\lfloor 1/b \rfloor (L-1)$ middle-stage nodes are inaccessible from y. Thus, there is at least one middle-stage node accessible from both x and y.

It remains to show that if $M = M^* - 1$, then the Clos network $C_{N,L,M}$ is not strictly nonblocking. The argument is similar to that in the proof of Theorem 1. □

With a minor change in the proof, it can be shown that Theorem 2 continues to hold if $B \in (1-b, 1]$. If $b = 0$ and B is arbitrary, then a straightforward analysis gives $M^* = M'$, where

$$M' = \lim_{\epsilon \downarrow 0} 2 \left\lfloor \frac{L-B}{1-B+\epsilon} \right\rfloor + 1$$

$$= \begin{cases} 2 \left\lfloor \dfrac{L-B}{1-B} \right\rfloor + 1 & \text{if } \dfrac{L-B}{1-B} \text{ is not an integer,} \\[2ex] 2 \left\lfloor \dfrac{L-B}{1-B} \right\rfloor - 1 & \text{if } \dfrac{L-B}{1-B} \text{ is an integer.} \end{cases}$$

Unfortunately, we have not been able to determine M^* for the case $0 < b < B \leq 1-b$. However, from Theorem 1 and the above observation, we have the following bounds.

COROLLARY 1. *Suppose B is an integer multiple of b and $1/b$ is an integer. Then for the continuous bandwidth case*

$$2 \left\lfloor \frac{L-B}{1-B+b} \right\rfloor + 1 \leq M^* \leq 2 \left\lfloor \frac{L-B}{1-B} \right\rfloor + 1.$$

We may be tempted to conjecture that $M^* = M'$ for the case where $0 < b < B \leq 1 - b$. However, this is not generally true, as can be seen by considering a Clos network with $L = 5$, $b = 0.1$, and $B = 0.8$. In this case $M' = 41$ and it can be shown that $M^* = 39$.

3.3. Comparison with the sufficient condition of Melen and Turner. Melen and Turner [1] obtained an upper bound for M^* for the continuous bandwidth case. We compare their upper bound with the results of §3.2. Assume throughout the discussion that $b \leq 1/2$.

For the case $B = 1$, the sufficient condition in [1] for $C_{N,L,M}$ to be strictly nonblocking is

$$M \geq 2 \left\lfloor \frac{L-1}{b} \right\rfloor + 3.$$

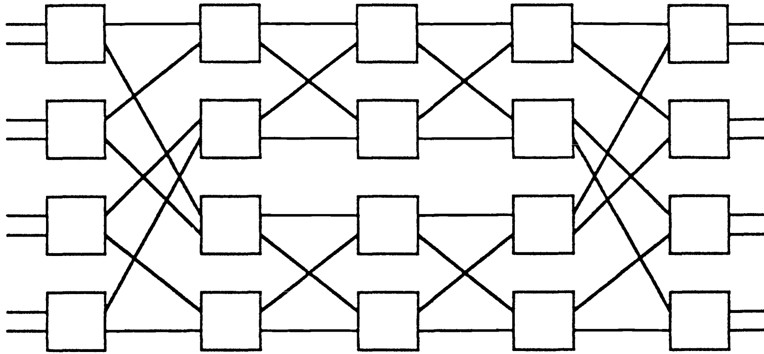


FIG. 2. *Beneš network* $B_{8,2}$.

(Put $\beta = 1$, $B = 1$, and $\omega = 1 - b$ in the formula in [1].) For the same case, the necessary and sufficient condition (see Theorem 2) is

$$M \geq 2\lfloor 1/b \rfloor (L-1) + 1 = M^*.$$

Thus, the sufficient condition in [1] calls for at least two unnecessary middle-stage nodes. For example, if $b = 0.255$ and $L = 5$, then the sufficient condition calls for 33 middle-stage nodes when only 25 are needed.

As another example, consider the case $B = b$ with $1/b \geq 2$ an integer. Then it follows from Theorem 1 that $M^* = 2(L-1) + 1$. The sufficient condition in [1] is

$$M \geq 2 \left\lfloor \frac{L-b}{1-b} \right\rfloor + 1,$$

which calls for (approximately) $2b(L-1)/(1-b)$ additional middle-stage nodes. In particular, if $B = b = 1/2$, then the number of middle-stage nodes called for by the sufficient condition is approximately twice that needed.

4. Cantor networks. A Beneš network with N inputs consisting of $L \times L$ square arrays (where $\log_L N$ is an integer) shall be denoted by $B_{N,L}$. Recall that (i) $B_{L,L}$ is simply an $L \times L$ square array; (ii) $B_{N,L}$ is constructed by stacking L Beneš networks $B_{N/L,L}$ on top of each other, adding a column of N/L square arrays of dimension $L \times L$ to both the input and output, and making appropriate connections (see [1], [3] for a more formal definition). Note that the Beneš network $B_{N,L}$ has $2H - 1$ stages, where $H := \log_L N$. The Beneš network $B_{8,2}$ is given in Fig. 2.

The Cantor network $K_{N,L,Q}$ can be constructed by stacking Q Beneš networks $B_{N,L}$; adding a column of N, $1 \times Q$ arrays at the input; adding a column of N, $Q \times 1$ arrays at the output; and making the appropriate connections. The Cantor network $K_{8,2,3}$ is given in Fig. 3.

Note that the Cantor network $K_{N,L,Q}$ has $2H + 1$ stages with stage $H + 1$ being the middle stage. Further note that between a given input port x and a given middle-stage node u there is exactly one path. Thus each input port x generates a directed tree with root node x and with the set of leaves being the nodes in the middle stage.

Let Q^* be the minimum Q such that $K_{N,L,Q}$ is strictly nonblocking. It is well known that $Q^* = \log_2 N$ for $L = 2$ for classical circuit switching [4]. In §§4.1–4.3 we determine M^* for the multirate model for both the discrete and continuous bandwidth cases.

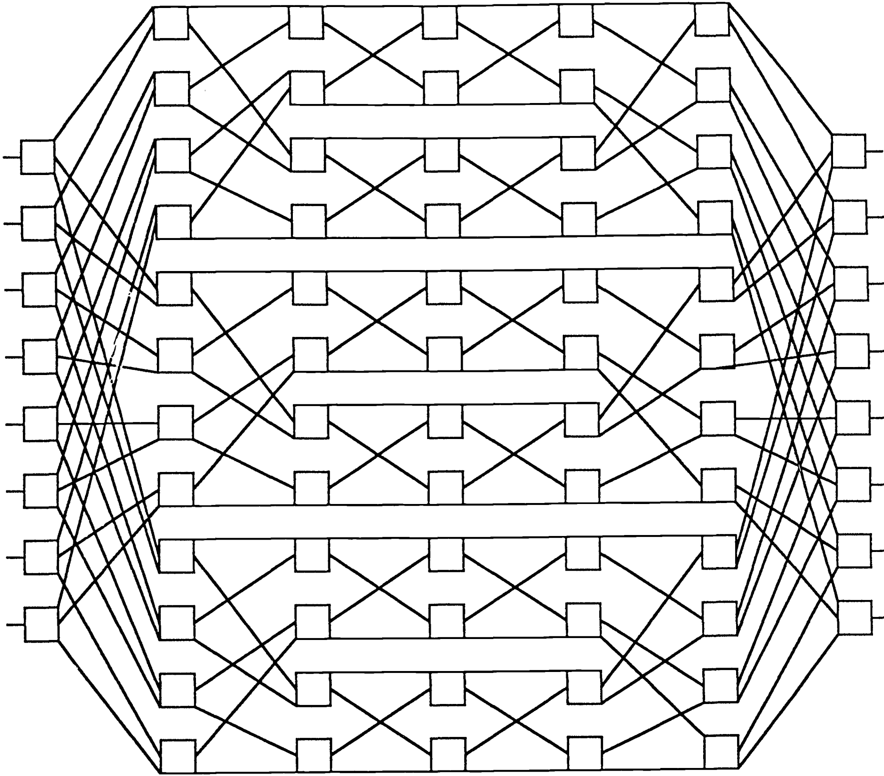


FIG. 3. *Cantor network* $K_{8,2,3}$.

4.1. The discrete bandwidth case. Let

$$s(\omega) := \sum_{h=2}^{H} L^{H-h} \left\{ \left\lfloor \frac{L^{h-1} - \omega}{1 - \omega + b} \right\rfloor - \left\lfloor \frac{L^{h-2} - \omega}{1 - \omega + b} \right\rfloor \right\}.$$

THEOREM 3. *For the discrete bandwidth case,*

$$Q^* = \left\lfloor 2 \frac{L}{N} s(B) \right\rfloor + 1. \tag{11}$$

Proof. We first show that K_{N,L,Q^*} is strictly nonblocking with Q^* given by (11). Suppose we want to establish a compatible connection (x, y, ω) to some existing configuration. Denote T for the directed tree generated by x, and denote T_h for the set of nodes in T that are in the hth stage, $h = 1, \cdots, H+1$. Note that a node in T is either accessible or inaccessible from x depending on the existing configuration. Say that a node $u \in T$ is inaccessible from x for the first time if (i) node u is inaccessible from x; and (ii) the predecessor of u in the tree T is accessible from x. Since there are L output links from each node in each stage, $h = 1, \cdots, H$, if a node in stage h is inaccessible for the first time, it will cause L^{H+1-h} middle-stage nodes to be inaccessible from x. Denote α_h for the number of nodes in stage h that are inaccessible from x for the first time, $h = 3, \cdots, H+1$. Then the number of middle-stage nodes that are inaccessible

from x is

$$\sum_{h=3}^{H+1} L^{H+1-h}\alpha_h.$$

Let P_j be the set of nodes in stage 1 such that there is a path in the original graph $G = (V, E)$ to some node in T_j. It is easily seen that $|P_j| = L^{j-1}$.

For a node $u \in T_j$ to be inaccessible for the first time, the incoming edge to u in T must have a weight $\geq 1 - \omega + b$. The total amount of weight available to edges in T incoming to the nodes in $\bigcup_{h=3}^{j} T_h$ is $|P_{j-1}| - \omega$. Thus,

$$\sum_{h=3}^{j} \alpha_h \leq \left\lfloor \frac{|P_{j-1}| - \omega}{1 - \omega + b} \right\rfloor, \qquad j = 3, \cdots, H+1.$$

Therefore, the number of inaccessible middle-stage nodes from x can be no more than $t(\omega)$, where

$$t(\omega) = \max \sum_{h=3}^{H+1} L^{H+1-h}\alpha_h$$

$$\text{s.t.} \ \sum_{h=3}^{j} \alpha_h \leq \left\lfloor \tfrac{L^{j-2}-\omega}{1-\omega+b} \right\rfloor, \qquad j = 3, \cdots, H+1,$$

$$\alpha_h \text{ integer}, \qquad h = 3, \cdots, H+1.$$

From polymatroid theory (see [2, Thm. 2, §18.4]) we know that the optimal solution to the above integer program is the greedy solution, namely,

$$\alpha_h = \left\lfloor \frac{L^{h-2} - \omega}{1 - \omega + b} \right\rfloor - \left\lfloor \frac{L^{h-3} - \omega}{1 - \omega + b} \right\rfloor, \qquad h = 3, \cdots, H+1.$$

Thus $t(\omega) = s(\omega)$. It is not difficult to show that $s(\omega) \leq s(B)$ for all $\omega \in [b, 1]$. Thus the maximum number of middle-stage nodes that are inaccessible from x is $s(B)$. Similarly, the maximum number of middle-stage nodes that are inaccessible from y is $s(B)$. Thus, there are at most $2s(B)$ inaccessible middle-stage nodes. Since K_{N,L,Q^*} has at least $2s(B) + 1$ middle-stage nodes, it follows that K_{N,L,Q^*} is strictly nonblocking.

It remains to show that if $Q = Q^* - 1$, then $K_{N,L,Q}$ is not strictly nonblocking. This is done by working the above argument backwards, as in the proof of Theorem 1. □

4.2. The continuous bandwidth case.

THEOREM 4. *For the continuous bandwidth case with* $B = 1$,

$$Q^* = \left\lfloor 2\lfloor 1/b \rfloor \frac{L-1}{L}(H-1) \right\rfloor + 1. \tag{12}$$

Proof. We first show that K_{N,L,Q^*} is strictly nonblocking with Q^* given by (12). Suppose we want to establish a compatible connection (x, y, ω) to some existing

configuration. As in the proof of Theorem 4, let α_h be the number of nodes in stage h that are inaccessible from x for the first time, $h = 3, \cdots, H+1$. Note that

$$\sum_{h=3}^{j} \alpha_h \leq J(\omega, L^{j-2}), \qquad j = 3, \cdots, H+1,$$

where $J(\cdot,\cdot)$ is defined in §3.2. Thus, the number of inaccessible middle-stage nodes can be no more than $t(\omega)$, where

$$t(\omega) := \max \sum_{h=3}^{H+1} L^{H+1-h} \alpha_h$$

$$\text{s.t.} \sum_{h=3}^{j} \alpha_h \leq J(\omega, L^{j-2}), \qquad j = 3, \cdots, H+1,$$

$$\alpha_h \text{ integer}, \qquad h = 3, \cdots, H+1.$$

Note that $J(\omega, L^{j-1}) \leq J(\omega, L^j)$, $j = 2, \cdots, H$. It therefore follows from polymatroid theory (see [2]) that the optimal solution to the above integer program is

$$\alpha_h = J(\omega, L^{h-2}) - J(\omega, L^{h-3}), \qquad h = 3, \cdots, H+1,$$

so that

$$t(\omega) = \sum_{h=3}^{H+1} L^{H+1-h} \left[J(\omega, L^{h-2}) - J(\omega, L^{h-3}) \right]. \tag{13}$$

From Lemma 1 we have

$$J(\omega, L^{h-2}) \leq J(1, L^{h-2}) = \lfloor 1/b \rfloor (L^{h-2} - 1), \qquad h = 2, \cdots, H+1. \tag{14}$$

Combining (13) and (14) gives

$$\begin{aligned} t(\omega) &\leq t(1) = \lfloor 1/b \rfloor \sum_{h=3}^{H+1} L^{H+1-h} (L^{h-2} - L^{h-3}) \\ &= \frac{N}{L} \lfloor 1/b \rfloor \frac{L-1}{L} (H-1). \end{aligned}$$

Since K_{N,L,Q^*} has at least $2t(1)+1$ middle-stage nodes it follows that K_{N,L,Q^*} is strictly nonblocking.

It remains to show that if $Q = Q^* - 1$, then $K_{N,L,Q}$ is not strictly nonblocking. This is done by working the above argument backwards as in the proof of Theorem 1. □

With a minor change in the proof, it can be shown that Theorem 4 continues to hold if $B \in (1-b, 1]$. If $b = 0$ and B is arbitrary, then it can be shown that $Q^* = Q'$, where

$$Q' = \lim_{\epsilon \downarrow 0} \left\lfloor 2 \sum_{h=2}^{H} L^{1-h} \left\{ \left\lfloor \frac{L^{h-1} - B}{1 - B + \epsilon} \right\rfloor - \left\lfloor \frac{L^{h-2} - B}{1 - B + \epsilon} \right\rfloor \right\} \right\rfloor + 1. \tag{15}$$

As in the case of the Clos network, we have not been able to determine Q^* for the case $0 < b < B \leq 1 - b$. However, from Theorem 3 and the above observation we have the following bounds.

COROLLARY 2. *Suppose B is an integer multiple of b and $1/b$ is an integer. Then for the continuous bandwidth case*

$$\left\lfloor 2\frac{L}{N}s(B)\right\rfloor + 1 \leq Q^* \leq Q'.$$

4.3. Comparison with the sufficient condition of Melen and Turner. For the continuous bandwidth case with $B = 1$, the sufficient condition of [1] for $K_{N,L,Q}$ to be strictly nonblocking is

$$Q \geq 2\frac{1}{Lb}(1 + (L-1)(H-1)).$$

For the same case, the necessary and sufficient condition (see Theorem 4) is

$$Q \geq \left\lfloor 2\lfloor 1/b\rfloor \frac{L-1}{L}(H-1)\right\rfloor + 1 = Q^*.$$

Thus, the sufficient condition of [1] calls for (approximately) $2\lfloor 1/b\rfloor(L-1)/L$ unnecessary Beneš networks $B_{N,L}$. For example, if $b = 0.255$, $L = 2$, and $N = 32$, then the sufficient condition calls for 20 Beneš networks when only 13 are needed.

For the case $b = B \leq \frac{1}{2}$ with $1/b$ an integer, the sufficient condition of [1] becomes

$$Q \geq 2\frac{1}{L(1-b)}(1 + (L-1)(H-1)).$$

For the same case, the necessary and sufficient condition is

$$Q \geq \left\lfloor 2\frac{L-1}{L}(H-1)\right\rfloor + 1 = Q^*.$$

In particular, if $B = b = \frac{1}{2}$, then the number of Beneš networks $B_{N,L}$ called for by the sufficient condition is (approximately) twice what is necessary.

5. Rearrangeable networks. Recall that for all N and L the Beneš network $B_{N,L}$ is rearrangeable for classical circuit switching (i.e., $b = 1$). Unfortunately the Beneš network is not generally rearrangeable when multirate traffic is present.

Counterexample 1. Consider $B_{8,2}$ supporting connections with weights 1 and b (with $b \leq \frac{1}{2}$). Suppose there are five compatible connections: $(1,1,1)$, $(3,3,1)$, $(5,4,1)$, $(4,2,6)$, $(6,2,6)$. It is easily seen that there does not exist a compatible configuration that realizes these connections.

Counterexample 2. Consider $B_{9,3}$ supporting connections with weights 1 and b (with $b \leq \frac{1}{2}$). Suppose there are seven compatible connections: $(1,1,1)$, $(2,2,1)$, $(4,4,1)$, $(5,5,1)$, $(7,6,1)$, $(6,3,6)$, $(9,3,6)$. Again, there does not exist a compatible connection that realizes these connections.

However, we do have the following positive result.

THEOREM 5. *Let G be a network that is rearrangeable for the classical circuit switching. Then G is also rearrangeable if all connections have the same weight b.*

Proof. Let C be a set of compatible connections. Construct a bipartite graph with one node for each input port, one node for each output port, and one edge from node x to node y for each connection (x, y, b) in C. Note that each node in the graph can have a degree of at most $1/b$. Therefore, by the graph coloring theorem (e.g., see [8]), we can color the edges in the graph with $1/b$ different colors so that no two connections involving the same input or output port are assigned the same color. Given the coloring, we route like colored connections so that no two share a common link (which we can do since the network is rearrangeable for classical circuit switching). We can do this for all colors and since each link can have at most one connection of each color, we are guaranteed not to have exceeded the capacity of any link. Thus, there is a compatible configuration that realizes C. □

Now suppose that all connections have weight of either b or 1. We know from Counterexamples 1 and 2 that Beneš network is not generally rearrangeable for this case. Thus, we need to consider networks that are more complex. To this end, consider a network G that is strictly nonblocking for classical circuit switching. For this network we can first route all connections with weight 1 to their destination ports. If we then remove all links that have a weight of 1, including those adjacent to input and output ports, the network remains strictly nonblocking for classical circuit switching, and hence rearrangeable for classical circuit switching. Thus, from Theorem 5 we can route all the connections of weight b along the remaining links. Summarizing, we have the following result.

COROLLARY 3. *Suppose G is strictly nonblocking for classical circuit switching. Then G is rearrangeable if all connections have weight of either b or* 1.

It would be of interest to show that Corollary 3 holds for the general discrete bandwidth case with K distinct rates. We have not succeeded in establishing this result, nor at constructing a counterexample. Now consider the network $K_{N,L,K}$, which contains K Beneš networks. Suppose that all connections of weight b_k are routed to the kth Beneš network. From Theorem 5 it follows that each of the Beneš networks can rearrange their single-rate connections. We therefore obtain the following result.

COROLLARY 4. *$K_{N,L,K}$ is rearrangeable for the discrete bandwidth case.*

Acknowledgment We thank the referee for his insightful observations.

REFERENCES

[1] R. MELEN AND J. S. TURNER, *Nonblocking multirate networks,* SIAM. J. Comput., 18 (1989), pp. 301–313.
[2] D. J. A. WELSH, *Matroid Theory,* Academic Press, London, 1976.
[3] V. E. BENEŠ, *Mathematical Theory of Connecting Networks and Telephone Traffic,* Academic Press, New York, 1965.
[4] D. G. CANTOR, *On non-blocking switching networks,* Networks, 1 (1971), pp. 367–377.
[5] C. CLOS, *A study of nonblocking switching networks,* Bell Systems Tech. J., 32 (1953), pp. 406–424.
[6] G. M. MASSON, G. C. GINGHER, S. NAKAMURA, *A sampler of circuit switching networks,* Computer J., (1979), pp. 145–161.
[7] C. SHANNON, *Memory requirements in a telephone exchange,* Bell Systems Tech. J., 29 (1950), pp. 343–349.
[8] M. GORDRAN AND M. MINOUX, *Graphs and Algorithms,* John Wiley, New York, 1984.
[9] G. NIESTEGGE *Nonblocking multirate switching networks,* in Proc. 5th ITC Seminar, Lake Como, Italy, May 1987

SIAM J. COMPUT.
Vol. 20, No. 4 , pp. 737–755, August 1991
© 1991 Society for Industrial and Applied Mathematics
010

INTERSECTING LINE SEGMENTS IN PARALLEL WITH AN OUTPUT-SENSITIVE NUMBER OF PROCESSORS*

MICHAEL T. GOODRICH†

Abstract. An efficient parallel algorithm is given for constructing the arrangement of n line segments in the plane, i.e., the planar graph determined by the segment endpoints and intersections. This algorithm is efficient relative to three efficiency measures—it is an NC algorithm, it has a small time-processor product, and it is output-size sensitive. In particular, it runs in $O(\log n)$ time using $O(n \log n + k)$ processors, where k is the size of the output (which is $\Omega(n^2)$ in the worst case). The algorithm does not receive the value of k as input, it determines it on-line. A method for solving an important special case of the segment arrangement problem is also shown, namely, when each input segment is parallel to one of the coordinate axes (i.e., iso-oriented). The algorithm for this problem runs in $O(\log n)$ time using an optimal $O(n + k/\log n)$ processors. The model of computation is the CREW PRAM model, where processor allocation must be explicit and global.

Key words. computational geometry, line-segment intersection, parallel algorithms, parallel data structures, PRAM model

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction.

One of the major thrusts of computational geometry research has been to show that we can solve many geometric construction problems with a running time that is proportional to the input size plus the output size (times logarithmic factors in some cases); see, for example, [6], [11], [12], [20], [25], [27], [31], [39]. This is significant, because most of these problems have trivial $\Omega(n^2)$ lower bounds, which are based on constructing examples that have a large output size. These worst-case examples seldom arise in practice, however. Thus, an algorithm whose running time is essentially linear in the size of the output will perform much better than the worst-case time on most inputs.

1.1. The problem.

One of the most studied of these problems is the problem of constructing the planar graph determined by the pairwise intersections of a set of line segments in the plane, i.e., the *segment arrangement* problem (see [6], [11], [12], [16], [30], [34]). This problem has several applications in computer graphics, for example, [21], [33], [37]. One of the oldest algorithms solving the segment arrangement problem is an elegant method by Bentley and Ottmann [6] published in 1979 that uses the now-famous "plane-sweeping" paradigm [16], [30], [34]. The running time of their algorithm is sensitive to the size of the output, as it runs in $O((n + k) \log n)$ time for the general case, and in $\Theta(n \log n + k)$ time if the input segments are iso-oriented (i.e., if each segment is parallel to one of the coordinate axes), where k is the size of the output. Since k is $\Omega(n^2)$ in the worst case, the existence of an optimal algorithm, running in $O(n \log n + k)$ time, became an open problem. This gave rise to a considerable amount of research done to resolve this question (e.g., [12], [13], [19]), and Chazelle and Edelsbrunner showed in 1988 that we can in fact solve this problem in $\Theta(n \log n + k)$ time.

* Received by the editors May 17, 1989; accepted for publication (in revised form) September 12, 1990. This research was announced in preliminary form in the Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, pp. 127–137. This research was supported by National Science Foundation grants CCR-8810568 and CCR-9003299, and National Science Foundation and Defense Advanced Research Projects Agency under grant CCR-8908092.

† Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218 .

In this paper we investigate how efficiently we can solve this problem in parallel. Our primary goal is to design a parallel algorithm that runs as fast as possible. Given that, our secondary goal is to design an algorithm that has a time-processor product that is as small as possible. Our motivation for this is that we desire an algorithm that can be simulated on a real machine, with a fixed constant number of processors, so as to maximize the speedup over the best-known sequential algorithms. The product of the time bound and processor bound characterizes the *work* that such a simulation would perform, and provides a simple measure of the algorithm's efficiency relative to the best known sequential algorithms. Thus, for the segment arrangement problem, we desire an algorithm that runs in $O(\log n)$ time and has an output-sensitive work bound.

1.2. Previous work. Prior to a preliminary announcement of this research [22], we knew of no previous work for solving this problem in parallel, other than the trivial brute-force method based on sorting that runs in $O(\log n)$ time using $O(n^2)$ processors (e.g., using Cole's sorting method [15]). The only known results were for solving special cases of the segment arrangement problem in parallel. For example, Atallah, Cole, and Goodrich [3] addressed the decision version of this problem, i.e., determining if *any* two segments intersect, deriving a method running in $O(\log n)$ time using $O(n)$ processors. In [14] Chow studied a restricted version of the problem: namely, she showed how to determine all the pairwise intersections of n iso-oriented segments. Her algorithm runs in $O((1/\epsilon)\log n + k_{\max})$ time using $O(n^{1+\epsilon})$ processors [14], where $\epsilon > 0$ is a small constant and $k_{\max}$ is the maximum, taken over all input segments s, of the number of intersections on s. Note that this does not give an NC algorithm, since $k_{\max}$ is $\Omega(n)$ in the worst case, nor does it balance the computational burden for the case when only a few segments cause the majority of intersections. Neither of these approaches seem to extend to the general segment arrangement problem.

Following the preliminary announcement of this research, however, there have been a number of results that apply to this problem. In particular, Anderson, Beame, and Brisson [2] and Hagerup, Jung, and Welzl [26] have studied the related problem of constructing the arrangement of n lines in the plane (which, of course, always has $\Theta(n^2)$ size), a problem that can be solved sequentially in $O(n^2)$ time [13], [17], [19]. The method of Anderson, Beame, and Brisson builds upon the methods presented in [22] to derive a parallel algorithm running in $O(\log n \log^* n)$ time using $O(n^2/\log n)$ processors in the CREW PRAM model. The method of Hagerup, Jung, and Welzl is a randomized method running in $O(\log n)$ expected time using $O(n^2/\log n)$ processors in the CRCW PRAM model. Subsequentially, Goodrich has improved upon these methods to derive a deterministic method running in $O(\log n)$ time using an optimal $O(n^2/\log n)$ processors in the CREW PRAM model [23], solving an open problem posed in the preliminary version of this paper [22]. Of course, if we apply these methods to the segment arrangement problem, then these methods are efficient only if k, the number of intersections, is large.

In addition to these algorithms for the line arrangement problem, Rüb (see [36]) has independently shown that one can solve the segment arrangement problem in $O(\log n \log\log n)$ time using $O(n + k)$ processors in the CREW PRAM model. Her method improves upon the line arrangement algorithms, then, for instances when k is not too large (e.g., $k << n^2/\log n \log\log n$).

1.3. Our results. The main result of this paper is an output-sensitive parallel algorithm for solving the segment arrangement problem. Our algorithm runs in $O(\log n)$ time using $O(n \log n + k)$ processors, where k is the size of the output. Note

that the work performed by our algorithm matches the time-processor product of the brute-force approach when the output size is large, i.e., when k is $\Omega(n^2)$, and is smaller than the method of Rüb for $k >> n \log n / \log \log n$. We also give an algorithm for the case when the segments are iso-oriented that runs in $O(\log n)$ time using an optimal $O(n + k/\log n)$ number of processors. Our model of computation is the CREW PRAM model, where processor allocation must be explicit and global.

The main obstacle to designing an output-sensitive parallel algorithm for the general segment arrangement problem is that paradigms that led to efficient sequential algorithms, such as plane-sweeping [16], [34], topological sweeping [12], [17], and incremental construction [16], [34], seem inherently sequential. Moreover, parallel techniques that worked well for parallelizing fast plane-sweeping algorithms, such as the plane-sweep tree [1], [3], cascading divide-and-conquer [3], and parallel sequence-evaluation [4], cannot be directly applied, for they require one to know a priori all the places where a sweeping line would need to stop. Such a requirement "begs the question" in the case of constructing a segment arrangement, for a sweep-line would need to stop at each intersection point.

Our algorithm, instead, is based on a number of new parallel algorithmic techniques, as well as a new geometric characterization of the types of intersections that can occur. The new parallel techniques include a "truncated" version of the zone lemma of [11]–[13], [18], and [19] and a method for reusing processors created for enumerating intersections of one type to then discover intersections of another type. The new geometric characterization is a "hierarchical" extension of a characterization due to Chazelle [11]. Our algorithm achieves its output-sensitivity by computing the size of the output while it is computing the answer, and dynamically allocates new processors accordingly. Our algorithm for the special case when the input segments are iso-oriented also uses this dynamic-allocation paradigm, in addition to the use of a "compressed" version of the array-of-trees parallel data structure of Atallah, Goodrich, and Kosaraju [4].

In the next section we discuss dynamic processor allocation in more detail, and show how to solve an important dominance reporting problem using this paradigm in §3. This problem arises as a natural subproblem in our segment arrangement algorithm, which we describe at a high level in §4. We give the details of our method in §§5 and 6. In §7 we present our algorithm for the iso-oriented case, and we conclude in §8.

2. A word about the computational model. The computational model we use in this paper is the Parallel Random Access Machine, or PRAM. Processors in this model act in a synchronous fashion and use a shared memory space. This model is divided into three types based on how memory can be accessed: the Exclusive-Read, Exclusive-Write (or EREW) model, the Concurrent-Read, Exclusive-Write (or CREW) model, and the Concurrent-Read, Concurrent-Write (or CRCW) model. All of our algorithms are for the CREW PRAM model.

Given an input of size n, the traditional way of utilizing this model is that we simply allocate, once and for all, a number of processors that depends on n (e.g., n^2, $n \log n$, etc.). Of course, a real parallel machine has a constant number of processors, c, not a number that is a function of n. Thus, the c real processors must simulate the "virtual" processors in the algorithm in order to implement it. Since we wish to solve a problem in an output-sensitive manner, in order to achieve the maximum speedup possible we allow the set of virtual processors to grow dynamically.

There are essentially two different ways to allow for a dynamically growing pool of

virtual processors. One approach, as outlined by Reif and Sen [35], is that of allowing a new virtual processor to be created by having some existing virtual processor execute a *spawning* operation. Such an operation is issued by an existing processor specifying the task that a new processor is to perform. Then, in the next time step, a new processor is created and begins executing that task. This is also similar to a model used by Bhatt and Cai [8]. This model does not specify how to implement the processor assignment should a number of different virtual processors simultaneously perform spawning operations, however.

The model we use in this paper does not allow for the spawning operation. Instead, we insist that for r new virtual processors to be allocated in time t we must have already constructed an r-element array that stores pointers to the r tasks these processors are to begin performing in step $t+1$. We refer to this as a *global allocation* scheme. This is essentially the same as the traditional PRAM model, in that every PRAM algorithm does such an allocation as its first step, usually to allocate a number of virtual processors that is a function of the input size.

It is beyond the scope of this paper to address all the relative strengths of the various dynamic processor allocation schemes. Nevertheless, we would like to mention that, in spite of its apparent weakness, the global allocation CREW PRAM model can simulate any algorithm designed for the spawning CREW PRAM model in a work-optimal fashion.

LEMMA 2.1. *If an algorithm A runs in t steps using p processors in the* CREW PRAM *model with local spawning of processors allowed, then A can be implemented in $O(t \log p)$ steps using $O(p/\log p)$ processors in the* CREW PRAM *model with global processor allocation.*

Proof. Let p_i denote the number of processors used in the spawning PRAM model in step i, and let T_i denote the list of tasks to be performed in step i, with $T_i[j]$ being the task to be performed by processor j. The main idea of the proof is to simulate step i of the spawning PRAM algorithm in $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors in the global-allocation PRAM model. We begin by performing all the nonspawning operations of step i. This can easily be done in $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors, by a simple application of Brent's theorem [10]. We then perform a parallel prefix computation[1] to determine $p_{i+1} - p_i$, the number of new processors that are to be spawned in step i, and to which tasks they are to be assigned. (Recall that a parallel prefix computation is one in which we reduce a problem to the problem of computing all prefix sums $s_k = \sum_{i=1}^{k} a_i$ of a list of numbers $(a_1, a_2, \cdots, a_n)$.) This gives us T_{i+1} and takes $O(\log p_i)$ time using $\lceil p_i/\log p_i \rceil$ processors [28], [29]. We complete the processing for step i by requesting $\lceil p_{i+1}/\log p_{i+1} \rceil - \lceil p_i/\log p_i \rceil$ new processors, bringing the total to $\lceil p_{i+1}/\log p_{i+1} \rceil$. This prepares us to simulate the next step in A. Thus, the entire algorithm can be implemented in $O(t \log p)$ time using $O(p/\log p)$ processors in the global-allocation PRAM model, where $p = p_t$. □

In the next section we illustrate the power of dynamic processor allocation by describing a simple, efficient parallel method for solving an important dominance reporting problem, which arises naturally in our segment arrangement algorithm.

[1] Recall that a parallel prefix computation is a reduction to the problem of computing all prefix sums $s_k = \sum_{i=1}^{k} a_i$ for n numbers $(a_1, a_2, \cdots, a_n)$, where $+$ is any associative operation. Also recall that this problem can be solved in $O(\log n)$ time using $O(n/\log n)$ processors [28], [29].

3. Dominance reporting. Suppose we are given two point sets A and B, consisting of n and m points, respectively. Moreover, suppose the points in A and B are sorted by increasing x-coordinates. We wish to construct, for each point p in B, a list that contains each point q in A such that $x(q) < x(p)$ and $y(q) < y(p)$, i.e., each point in A that p *dominates*. We let $\mathrm{Dom}(A,p)$ denote this set, and refer to this problem as the *two-set dominance reporting* problem.

We do not know of any previous work for this problem. Atallah, Cole, and Goodrich [3] address the counting version of this problem (where one is simply interested in determining the value of $|\mathrm{Dom}(A,p)|$, the number of points of A that p dominates), deriving an algorithm that runs in $O(\log N)$ time using $O(N)$ processors, where $N = \max\{n,m\}$. In this section we show how to construct $\mathrm{Dom}(A,p)$ for each p in B in $O(\log N)$ time using $O(N/\log N + l)$ processors, where l is the total number of answers ($l = \sum_{p \in B} |\mathrm{Dom}(A,p)|$). Our method uses a different approach than that taken by Atallah, Cole, and Goodrich.

3.1. A simple data structuring approach. We first describe a solution based on the use of a simple data structure and global processor allocation. This method runs in $O(\log N)$ using $O(N+l)$ processors. We then show how to reduce the number of processors to $O(N/\log N + l)$ by some processing steps.

The approach is to build a data structure for the points of A and then query this structure for each point in B in parallel. In particular, the data structure, D, consists of a complete binary tree T with the points of A stored in its leaves in left-to-right order. Let v be an internal node of T; and let z, u, and w be, respectively, the parent, left child, and right child of v. For each such v we store a list $A(v)$, which contains all the points stored in descendants of v sorted by their y-coordinates. In addition, we augment each element p of $A(v)$ with pointers to p's predecessor in $A(z)$, $A(u)$, and $A(w)$ (recall that p's predecessor in a list $A(*)$ is the largest element in $A(*)$ smaller than p), using y-coordinates as comparison keys. Such a structure is easily constructed by Cole's parallel mergesort method [15] in $O(\log N)$ time using $O(N)$ processors.

We then perform two queries for each point p in B. The first query is to determine the size of $\mathrm{Dom}(A,p)$, and the second query is to construct $\mathrm{Dom}(A,p)$. The first query is answered by searching for the leaf position of x in T, starting at the root, while simultaneously locating the position of y in each $A(v)$ list such that v is on the left fringe of the search path (i.e., v is the left child of a node on the search path but is, itself, not on the search path). The elements less than y in each such $A(v)$ constitute the set of answers for p. Thus, we can perform this counting query for any p in $O(\log N)$ time, using a single processor, simply by adding up the ranks of the predecessor of p in each of these lists. Given the sizes of all the $\mathrm{Dom}(A,p)$ lists we can then perform a parallel prefix computation to determine the total number of answers and allocate the space for a global array Dom that will store all the $\mathrm{Dom}(A,p)$ lists as subarrays. We can then perform a global allocation of l processors that then collectively enumerate the answers in $O(\log N)$ time, filling in all the "slots" in the Dom array. Thus, the total procedure can be implemented in $O(\log N)$ time using $O(N+l)$ processors.

3.2. Improving the processor bounds. The method described above suffers from two inefficiencies: (1) it builds the data structure D using every point in A, including points that will not be included in any $\mathrm{Dom}(A,p)$ list, and (2) it performs a dominance reporting query for each point p in B, even if $\mathrm{Dom}(A,p)$ may turn out

to be empty. We can easily remove both of these inefficiencies by performing the following preprocessing steps, however:

(1) In this step we remove from B each point p such that $\mathrm{Dom}(A,p)$ is empty. We can determine, for each p in B, whether or not $\mathrm{Dom}(A,p)$ is empty by performing a parallel prefix computation on A to determine, for each q in A, the value $\mathrm{Min}Y(q) = \min_{q' \in A}\{y(q') \colon x(q') \leq x(q)\}$, and then performing a merge of A and B by increasing x-coordinates. Both of these operations, of course, take advantage of A and B being presorted. For any point p in B, it is easy to see that $\mathrm{Dom}(A,p) \neq \emptyset$ if and only if $\mathrm{Min}Y(q) < y(p)$, where q is the immediate predecessor of p in A. Given the merging of A and B, we can easily test this condition for each p in B in $O(1)$ time, and then perform a parallel prefix data compression procedure to remove any p's from B such that $\mathrm{Dom}(A,p) = \emptyset$. This step can be easily implemented in $O(\log N)$ time using $O(N/\log N)$ processors [9], [28], [29], [38].

(2) In this step we remove from A each point q that is not contained in any $\mathrm{Dom}(A,p)$ list. We do this using a method very similar to that used in Step 1. The time and processor bounds are as in Step 1.

Clearly, the total number of remaining points in A and B is dominated by l, the number of answers. Thus, by following this preprocessing step by the dominance reporting procedure described in the previous subsection, we derive the following lemma.

LEMMA 3.1. *Given two point sets A and B, with n and m points, respectively, sorted by increasing x-coordinates, we can construct* $\mathrm{Dom}(A,p)$ *for each p in B in $O(\log N)$ time using $O(N/\log N + l)$ processors in the* CREW PRAM *model, where $N = n + m$ and $l = \sum_{p \in B} |\mathrm{Dom}(A,p)|$.*

We make considerable use of this lemma in our method for constructing the arrangement of a collection of line segments. In fact, we usually need to solve a collection of 2-set dominance reporting problems in parallel. This presents no real problems, however, as we show in the following lemma.

LEMMA 3.2. *Given h instances of the 2-set dominance reporting problem, specified by h pairs of points sets (A_1, B_1), (A_2, B_2), $\cdots$, (A_h, B_h), we can construct* $\mathrm{Dom}(A_i,p)$ *for each point $p \in B_i$, $i = 1, \cdots, h$, in $O(\log N)$ time using $O(N/\log N + l)$ processors in the* CREW PRAM *model, where $N = \sum_{i=1}^{h} |A_i| + |B_i|$ and $l = \sum_{i=1}^{h} \sum_{p \in B_i} |\mathrm{Dom}(A_i,p)|$.*

Proof. The proof follows from a straightforward implementation of the method used to prove Lemma 3.1. The only modification necessary is that each place in the algorithm where a parallel prefix computation is performed (as a precursor to an allocation of new virtual processors), we must now coordinate h simultaneous parallel prefix computations. This is due to the requirement that dynamic processor allocation be global. As it does not raise any real difficulties, we leave the details of this implementation to the reader. □

Having discussed our computational model, and how it can be used for dominance reporting, we now give an overview of our method for constructing the arrangement of a collection of line segments.

4. An overview of our algorithm. Suppose we are given a set S of n line segments in the plane. We define the *upper* (respectively, *lower*) *vertical shadow* in S of a point p to be the point on the first segment in S that is intersected by the vertical ray emanating upward (respectively, downward) from p, if such a point exists. The *segment arrangement* of S is defined to be the planar graph determined by the pairwise

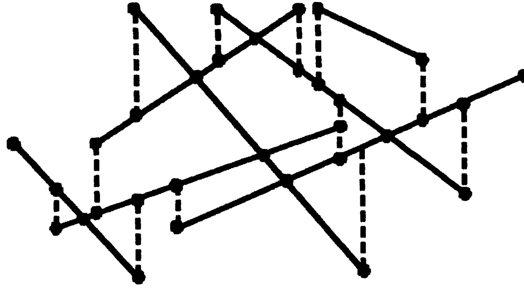


FIG. 1. *An example segment arrangement.*

intersections in S as well as all the vertical shadows of the endpoints of segments in S (see Fig. 1). The edges in this graph are determined by adjacent intersections (along some segment s) and by segment endpoints and their vertical shadows. For simplicity, we assume that at most two segments meet at any intersection point. We can easily modify our method to allow for multiple segments intersecting in the same point (using an appropriate definition of the "multiplicity" of an intersection point).

4.1. Characterizing intersections. Before we give our algorithm overview, we review an observation by Chazelle [11] for characterizing segment intersections in terms of a segment tree data structure [7]. Let T be the complete binary tree whose at most $2n + 1$ leaves, in left-to-right order, correspond to the regions, called *slabs*, determined by placing a vertical line through each endpoint of each segment in S. For each v in T we use Π_v to denote the union of all the slabs associated with the descendents of v (including v itself, if v is a leaf). A segment s_i *spans* a slab Π_v if s intersects both the left and right boundary of Π_v. A segment s_i *covers* a node $v \in T$ if it spans Π_v but not $\Pi_{\text{parent}(v)}$. Clearly, no segment covers more than two nodes on any level of T; hence, each segment covers at most $O(\log n)$ nodes of T. A segment s_i *ends in* Π_v if s_i does not span Π_v, but has an endpoint in Π_v. For each node $v \in T$ we define the following sets (see Fig. 2):

$$\text{Cover}(v) = \{s \in S \mid s \text{ covers } v\},$$
$$\text{End}(v) = \{s \in S \mid s \text{ ends in } \Pi_v\}.$$

We can characterize the intersections in S as follows.

OBSERVATION 4.1 ([11]). Let S be a set of line segments in the plane, and let s_1 and s_2 be two segments in S that intersect at a point p. In addition, let T be a segment tree for S. Then there is a (unique) node $v \in T$ such that $p \in \Pi_v$ and one of the following is true:

(1) $s_1, s_2 \in \text{Cover}(v)$,
(2) $s_1 \in \text{End}(v)$ and $s_2 \in \text{Cover}(v)$,
(3) $s_2 \in \text{End}(v)$ and $s_1 \in \text{Cover}(v)$.

We call intersections of type 1 CC-*intersections* and intersections of types 2 and 3 EC-*intersections.*

4.2. The method. Our method, which we describe below, is based on finding all the CC-intersections first, and then using those intersections to help determine all

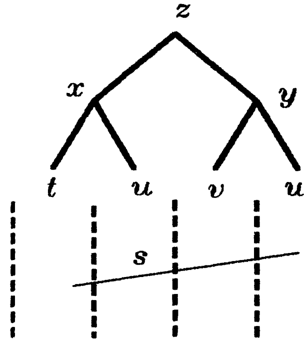


FIG. 2. *The segment s is in* Cover(u) *and* Cover(v), *as well as* End(t), End(w), End(x), End(y), *and* End(z).

the EC-intersections.

Step 1. In this step we construct a segment tree T for the segments in S, including the lists End(v) and Cover(v) for each $v \in T$. In addition, for each v in T, we sort the segments in Cover(v), where comparisons are based on the y-coordinates of the intersections of the segments with the left boundary of Π_v. This step can be easily implemented in $O(\log n)$ time using $O(n \log n)$ processors in the CREW PRAM model, by using the method of Aggarwal et al. [1] to construct T and the method of Cole [15] to sort each Cover(v) list (since the total size of all the Cover(v)'s is $O(n \log n)$).

Step 2. In this step we determine all the CC-intersections in S. Our method is based on the simple observation that if two segments in Cover(v) intersect, then their relative order would be reversed if we were to base comparisons on segment intersections along Π_v's right boundary rather than basing comparisons on segments intersections along Π_v's left boundary. We implement this step via a reduction to the dominance reporting problem, constructing, for each v in parallel, and for each segment s in Cover(v), a list of the other segments in Cover(v) that intersect s. We use these lists to construct the arrangement of the segments in Cover(v), which, following the convention of [11] and [12], we call the *hammock*. This step requires $O(\log n)$ time using $O(n \log n + \alpha)$ processors, where α is the total number of CC-intersections in S.

Step 3. In this our most involved step we compute all the EC-intersections in S. We implement it in two phases. In the first phase we find, for each $s \in$ End(v), all the EC-intersections of s with segments in Cover(v), so long as there are fewer than $c \log n$ such intersections (c is a constant parameter), or, alternatively, we determine if there are at least $c \log n$ such intersections. This requires $O(\log n)$ time using a processor per segment in End(v), for all v in T, and is based on a "truncated" version of the zone lemma of [11]–[13], [18], and [19]. In the second phase, then, we find, for each $s \in$ End(v), all the EC-intersections of s with segments in Cover(v), provided s has at least $c \log n$ such intersections. We restrict this phase to such segments, because our second phase requires at least $O(\log n)$ processors for each segment involved, and we wish to "charge" the cost of these processors to the intersections found. Our method runs in $O(\log n)$ time and takes advantage of a characterization similar to that of Observation 4.1. We conclude the construction by determining all the adjacencies between the intersection points and endpoints in the segment arrangement. This entire step requires $O(\log n)$ time using $O(n \log n + \alpha + \beta)$ processors, where β is the

number of EC-intersections in S.

So, assuming we can implement each of the above steps in the stated bounds, then we can enumerate all the pairwise intersections in S in $O(\log n)$ time using $O(n \log n + k)$ processors, where $k = \alpha + \beta$ is the size of the output. Let us now give the details for performing each of the above steps. The details for Step 1 should already be apparent, so we begin our detailed description with Step 2.

5. Computing CC-intersections.

In Step 2 we compute all the CC-intersections in S. Let us concentrate on the problem of finding all the CC-intersections for a specific node v in T; we perform this computation for each v in parallel. Recall that in Step 1 we constructed all the Cover(v) lists for the nodes in T. For each segment s in Cover(v), let $y_1(s)$ (respectively, $y_2(s)$) denote the y-coordinate of the intersection of s with the left (respectively, right) boundary of Π_v. The following observation characterizes all CC-intersections in terms of these labels.

OBSERVATION 5.1. Two segments r and s in Cover(v) have a CC-intersection intersection in Π_v if and only if one of the following is true:

(1) $y_1(r) < y_1(s)$ but $y_2(r) > y_2(s)$,

(2) $y_1(r) > y_1(s)$ but $y_2(r) < y_2(s)$.

For each segment s in Cover(v), if we define a point $p_s = (y_1(s), y_2(s))$, then we can interpret Observation 5.1 in terms of dominance relationships. Namely, a segment r has a CC-intersection with s if and only if p_r is (i) above and to the left of p_s, or (ii) below and to the right of p_s. Thus, determining all CC-intersections in some slab Π_v can be reduced to two instances of the 2-set dominance reporting problem, where the set Cover(v) plays the roles of both sets A and B of Lemma 3.2. Of course, we must also re-orient the x- and y-axes so that the dominance relation of interest is downward and to the left. Note that the condition of Lemma 3.1 requiring that the points in A and B be presorted by their first coordinates is immediately satisfied, since the segments in each Cover(v) list are sorted by the y-coordinates of their intersections with the left boundary of Π_v. Of course, since we must implement this step for all nodes v in parallel, we must apply Lemma 3.2. Therefore, since the total size of all the Cover(v) lists is $O(n \log n)$, this entire computation can be implemented in $O(\log n)$ time using $O(n + \alpha)$ processors, where α is the total number of CC-intersections.

5.1. Constructing the hammock.

To complete Step 2 we have only to construct the adjacency information for the hammock. That is, for each intersection point p of segments r and s we must determine the other intersection points on r and s, respectively, to which p is adjacent. We do this by sorting, for each s in parallel, the intersections along s (which were just computed) by x-coordinates. Then for each intersection point p of a segment s with a segment r we locate the position of p in the list for r by a binary search. From this we then construct a representation of the planar graph induced by the adjacencies of the CC-intersections for Cover(v) (e.g., [5], [24], [32], [34]). We finish the construction by augmenting the graph, as Chazelle does [11], by adding two pointers for each edge e that point to the leftmost and rightmost vertex, respectively, of each face in the hammock to which e belongs. Since this computation requires the sorting of $O(n \log n + \alpha)$ elements, it takes $O(\log n)$ time using $O(n \log n + \alpha)$ processors [15], which dominates the complexity of Step 2.

Thus, we have shown how to efficiently find all the CC-intersections in S and construct the hammock for each Cover(v) list. In the next section we address the problem of finding the EC-intersections in S.

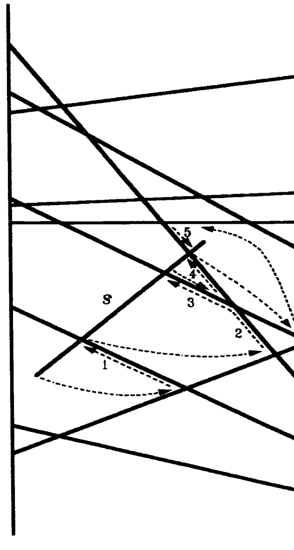


FIG. 3. *An example walk in the hammock. The traversed edges are numbered in the order they would appear in the walk in the hammock for s.*

6. Computing EC-intersections. To complete the algorithm we must implement Step 3, the finding of all the EC-intersections for each v in T. As mentioned earlier, this is the most involved step in the construction. It consists of two phases: one that finds the intersections along segments that have few EC-intersections, and the other that finds the intersections along segments that have many EC-intersections.

6.1. Segments with few intersections. Let us concentrate on the computations for a particular v in T. We begin by constructing a planar point location data structure for the hammock for v, e.g., using the method of Atallah, Cole, and Goodrich [3], which takes $O(\log n)$ time using $O(|\text{Cover}(v)| + \alpha_v)$ processors, where α_v is the number of CC-intersection determined by the segments in Cover(v). This requires $O(n \log n + \alpha)$ processors for all $v \in T$, and allows point locations to be performed in the hammock for a particular Cover(v) in $O(\log n)$ time using a single processor.

Suppose we are given a query segment s in End(v). We wish to find all the EC-intersections between s and segments in Cover(v), so long as there are fewer than $c \log n$ such intersections (where $c \geq 1$ is a constant parameter). We use the point location structure for Cover(v) to locate the two faces f_a and f_b that contain s's two endpoints a and b, respectively (with a being to the left of b). We then mimic the method of Chazelle [11] for walking through the hammock from f_a to f_b, except that we cut the walk short as soon as it traverses $4c \log n$ edges. We show below that if the walk is terminated early because of this restriction, then s must have at least $c \log n$ intersections with segments in Cover(v).

So let us review the method of Chazelle [11]. If $f_a = f_b$, then we are done, so let us assume $f_a \neq f_b$. We begin by jumping to the rightmost vertex v_1 in $f_1 = f_a$. We then traverse the edges of f_1 until we find the edge e_1 of f_1 that intersects s. If v_1 is above the line supporting s, then this traversal is clockwise, and is counterclockwise, otherwise. Upon reaching e_1, we use the adjacency information for e_1 to "hop" over

e_1 into the next face, f_2, which is adjacent to s. We then use the extra pointer for e_1 to jump to the rightmost vertex v_2 in f_2, going from face to face along s, provided that for each edge e traversed, the line supporting e intersects s. (See Fig. 3.) If we are about to traverse an edge whose supporting line does not intersect s, then we suspend the traversal from f_a at this point, and begin a symmetric traversal from f_b (using the rule that if v_i is above the line supporting s, then the traversal must be counterclockwise, and must be clockwise, otherwise). We continue this traversal until all the intersections along s have been discovered or, as in our case, we traverse at least $4c \log n$ edges. Chazelle [11] proves an important "zone" lemma for his scheme, establishing that if one uses his search strategy (without our extra stopping criterion, of course), then one will eventually discover all the intersections along s and the total time spent will be proportional to the number of intersections. The next lemma establishes a "truncated" version of this zone property.

LEMMA 6.1. *Suppose we have traversed at least 4δ edges in performing the walk for a segment s. Then there are at least δ intersections along s in the hammock.*

Proof. Since this is a slightly stronger version of a lemma proved by Chazelle [11], we use the proof technique of Chazelle, Guibas, and Lee [13] to prove it. Namely, we use an accounting scheme, where for each edge traversed, we charge one of the intersections along s for the cost of this traversal. Let f be a face traversed, and let s_i be the subsegment of s contained in f. The traversed edges of f can be divided into three groups: *left-hanging* edges, which intersect s left of s_i, *right-hanging* edges, which intersect s right of s_i, and *anchored* edges, which are adjacent to s_i. These groups suffice, because the line supporting each traversed edge intersects s and each f is convex. Hence, for any face f, all the nonanchored edges we traverse in f will be either left-hanging or right-hanging, but not both. The accounting scheme is that each left-hanging edge e charges the intersection of s with the line supporting e's successor in a clockwise traversal around f, and each right-hanging edge e charges the intersection of s with the line supporting e's successor in a counterclockwise traversal around f. Each anchored edge e simply charges its intersection with s. It is easy to see that each intersection point can be charged by at most one left-hanging edge, one right-hanging edge, and at most twice by its anchored edge. So each intersection point can be charged at most four times. Therefore, if we have traversed at least 4δ edges, then we must have charged at least δ intersection points. □

Thus, by this truncated zone lemma, if in traversing the hammock for a segment s we stopped by reaching the other endpoint of s, then we have discovered all the EC-intersections for s; and if we terminated the traversal early, then there must be at least $c \log n$ intersections of s with segments in Cover(v). Note, however, that the $c \log n$ intersection points need not be consecutive intersections along s.

Let E_v be the list of all segments in End(v) that have at least $c \log n$ EC-intersections, and let S_v denote the set of segment "pieces" in the hammock for v, i.e., the segments resulting from cutting each s in Cover(v) at its CC-intersections. Note that $\sum_{v\in T} |E_v|$ is at most $O(n \log n)$ and $\sum_{v \in T} |S_v|$ is at most $O(n \log n + \alpha)$.

6.2. Segments with many intersections. We have yet to find all the EC-intersections for the segments in E_v. Our method resembles a "recursive" application of the first two steps in our algorithm. Let us, then, concentrate on the computation for a specific node v in T, with the understanding that we perform this computation for all v in T in parallel.

We begin by building a segment tree T_v for the segments in S_v. To avoid confusion, let us denote the sets and slabs for each node w in T_v using lowercase letters. Thus,

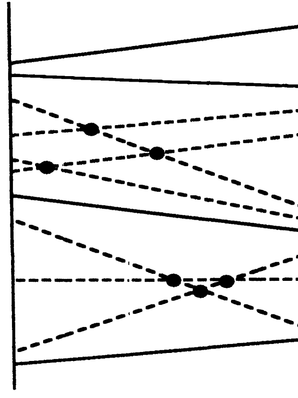


FIG. 4. *An example π_w. The segments in* end(w) *are shown dotted and the segments in* cover(v) *are shown solid.*

for each w in T_v we define lists cover(w) and end(w) in terms of the slab π_w associated with w. (See Fig. 4.) For each w in T_v we have cover(w) stored in sorted order by the segment intersections with the left vertical boundary of π_w. Let us also define a list left(w), which consists of all segments in end(w) that intersect the left boundary of π_w, and let us also store the left(w) lists sorted by the segment intersections with the left vertical boundary. Since the subsegments in S_v do not intersect, except at their endpoints, we can use the method of Atallah, Cole, and Goodrich [3] to build T_v. We use this method because it gives us the left(w) lists in sorted order without our having to explicitly perform a sorting operation. Note: the tree in the Atallah, Cole, and Goodrich construction is built on every $\lceil \log n \rceil$th x-coordinate; so that the end(w) list stored in a leaf has $O(\log n)$ size rather than $O(1)$ size. This will not affect the running time of our implementation by more than a constant factor, however. Their method runs in $O(\log m)$ time using $O(m)$ processors, where m is the number of segments. In our case $m = |\text{Cover}(v)| + \alpha_v$. Thus, we can use the processors created in Step 2 (to enumerate CC-intersections) to now help construct T_v for each v in T in parallel. This requires $O(\log n)$ time using a total of $O(n \log n + \alpha)$ processors.

For each w in T, we let inter(w) denote the set of segments in Cover(v) that have an intersection point in π_w. Recall that the segments in S_v are all pieces of segments in Cover(v) that span Π_v. We exploit this property to characterize EC-intersections in the following lemma, in a manner analogous to that of Observation 4.1.

LEMMA 6.2. *Given a node v in T, let s be a segment in E_v and t be a segment in* Cover(v), *and suppose s and t intersect at a point p. In addition, let T_v and S_v be as above, and let $\hat{t}$ be the portion of t in S_v that contains p. Then there is a (unique) node $w \in T_v$ such that $p \in \pi_w$ and one of the following is true:*

(1) $\hat{t} \in$ cover(w) *(a "type 1" intersection),*

(2) $t \in$ inter(w) *and s covers w (a "type 2" intersection),*

(3) $t \in$ inter(w) *and s ends in π_w, where w is a leaf (a "type 3" intersection).*

Proof. Let z be the leaf in T_v that contains p. There are two cases:

(1) s ends in π_z. If $\hat{t}$ does not span z, then $t \in$ inter(z); hence, p is a type 3 intersection. If $\hat{t}$ spans z, then there must be an ancestor w of z that $\hat{t}$ covers; hence, p is a type 1 intersection.

(2) s spans z. Let w be the ancestor of z that s covers. If $\hat{t}$ also covers w, then p is again a type 1 intersection. Otherwise, if $\hat{t}$ has an endpoint in π_w, then $t \in$ inter(w); hence, p is a type 2 intersection. □

We implement Step 3, then, by searching for each type of intersection.

Type 1 *intersections.* For each segment s in E_v, we allocate $O(\log n)$ processors to s and perform the following query at each node w in T_v such that s has an endpoint in π_w or s covers w:

> We locate the two endpoints of the segment $s \cap \pi_w$ (i.e., s "clipped" to π_w) in cover(w), by two binary searches. Note that this is possible, because the segments in cover(w) do not intersect, hence, are linearly ordered by the "above" relationship. All the segments in cover(w) between these two positions in the list must intersect s.

After performing this query, each processor assigned to s has determined some number of type 1 intersections for s, and, in fact, has an implicit representation of a list of these intersections. By performing a parallel prefix computation, then, we can allocate enough processors to enumerate all these type 1 intersections. This can easily be done in $O(\log n)$ time using $O(\sum_{v \in T} |E_v| \log n + \beta_1)$ processors (for all v in T), where β_1 is the total number of type 1 intersections.

Type 2 *intersections.* Our method is based on the observation that a type 2 intersection between $s \in E_v$ and $t \in \text{Cover}(v)$ is determined by a node w in T_v such that both s and t span π_w. Therefore, we can determine all such type 2 intersections by a reduction to the 2-set dominance reporting problem. In particular, we determine, for each node w in T_v, the set $ec(w)$ containing each segment $s \in E_v$ such that s covers w. We also sort each $ec(w)$ list by the y-coordinates of the points formed by the intersections of the segments in $ec(w)$ and the left boundary of π_w. This takes $O(\log n)$ time using $O(|E_v| \log n)$ processors. Note that the list left(w) stores a piece of each segment in inter(w), and these pieces are sorted by the y-coordinates of the points formed by the intersections of the segments in inter(w) and the left boundary of π_w. We associate a pair $(y_1(s), y_2(s))$ with each segment s in $ec(w)$ (respectively, left(w)), where $y_1(s)$ (respectively, $y_2(s)$) is the y-coordinate of the intersection of the left (respectively, right) vertical boundary of π_w with s. Thus, if interpreted as points, the elements of $ec(w)$ and left(w) are sorted by their first coordinates, satisfying the ordering precondition of Lemma 3.1. Just as in our method of §5, a solution to two instances of the 2-set dominance reporting problem gives us all the intersections between the "points" in $ec(w)$ and left(w) for each w in T_v. By Lemma 3.2 this takes $O(\log n)$ time using $O(n \log n + \sum_{v \in T} |E_v| + \alpha + \beta_2)$ processors (for all v in T), where β_2 is the number of type 2 intersections, since the total size of all $ec(w)$ lists is at most $O(\sum_{v \in T} |E_v| \log n)$ and the total size of all the left(w) lists is at most $O(n \log^2 n + \alpha \log n)$. Thus, the total number of processors needed for this step is $O(n \log n + \sum_{v \in T} |E_v| \log n + \alpha + \beta_2)$.

Type 3 *intersections.* Each type 3 intersection is determined by a leaf node w in T_v. Since $|\text{inter}(w)|$ in this case is $O(\log n)$, we can find all type 3 intersections by assigning a processor to each segment s in E_v and visiting each node w in T such that s ends in π_w. This processor simply tests each segment t with a piece $\hat{t}$ in end(w) to see if t intersects s. This clearly takes $O(\log n)$ time using $O(|E_v|)$ processors.

Having determined all three types of intersections completes the computation of the EC-intersections, giving us all the pairwise intersections of the segments in S. The total time needed is clearly $O(\log n)$. The total number of processors needed is $O(n \log n + \sum_{v \in T} |E_v| \log n + \alpha + \beta)$, where β is the number of EC-intersections. By construction, however, each segment s in E_v determines at least $c \log n$ EC-intersections; hence, $\sum_{v \in T} |E_v| \log n$ is $O(\beta)$. Therefore, the total number of processors needed is $O(n \log n + \alpha + \beta)$.

We complete our algorithm by constructing the segment arrangement, without vertical shadows, from the intersection points and endpoints, using essentially the same method we used to construct the hammocks (i.e., by sorting the intersections along each segment). We then augment this structure with the vertical shadows by applying the trapezoidal decomposition algorithm of Atallah, Cole, and Goodrich [3] and the sorting algorithm of Cole [15]. This takes $O(\log n)$ time using $O(n + k)$ processors, where $k = \alpha + \beta$. We summarize as follows.

THEOREM 6.3. *Given a set S of n line segments in the plane, we can construct the segment arrangement for S in $O(\log n)$ time using $O(n \log n + k)$ processors in the* CREW PRAM *model, where k is the size of the output.*

Thus, one can construct a segment arrangement efficiently in parallel in an output-sensitive manner. In the next section we show how to perform this construction optimally for the important special case when the segments are iso-oriented.

7. Iso-oriented segments. In this section we show how to construct the segment arrangement when all the segments are parallel to the x- or y-axes. Our method runs in $O(\log n)$ time using $O(n + k/\log n)$ processors in the CREW PRAM model, which is optimal. Since our algorithm is based on a "compressed" version of the *array-of-trees* parallel data structure of Atallah, Goodrich, and Kosaraju [4], we begin by reviewing this structure.

7.1. The array-of-trees. Suppose we are given a sequence $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_n)$, of **insert**$(a)$ and **delete**(a) operations. Let a_t denote the argument of the operation σ_t, and let A be the list of all distinct a_t values stored in sorted order. Also let A_t denote the set of items from A that would be present at "time" t if the operations $(\sigma_1, \cdots, \sigma_t)$ were evaluated sequentially, assuming that the initial set is $\emptyset$. A *tree query* is any query operation that can be performed on a complete binary tree T with $O(n)$ nodes in $O(\log n)$ time assuming that elements are stored in the leaves of T and each internal node v of T can store the values of $O(1)$ functions applied to values stored in v's children. Examples of such tree queries include the computation of the maximum y-coordinate of v's descendents or the computation of the number of v's descendents. The array-of-trees data structure allows us to perform any tree query on any given A_t in $O(\log n)$ time, assuming all the elements of A_t were stored in the leaves of a complete binary tree T. In fact, this structure can be viewed as an array of trees $(T_1, T_2, \cdots, T_n)$, where T_t is a complete binary tree whose leaves correspond to the elements of A, one element per leaf, such that the leaves associated with elements of A_t are *active* while all others are *in-active* (i.e, they store the **nil** value).

The "skeleton" of the array-of-trees is a complete binary tree T whose leaves are associated with the elements in A, one per leaf. For each a in A we construct $\sigma(a)$, the subsequence of σ consisting of all operations that have a as their argument. Note: with each operation in $\sigma(a)$ we store its position in σ; in fact, each time we refer to a σ_t, t denotes its index in σ. Using parallel sorting [15], it is easy to construct A, T, and all the $\sigma(a)$'s in $O(\log n)$ time using $O(n)$ processors.

For each v in T we construct a list $B(v)$ of records $(R_1, R_2, \cdots, R_{l_v})$ such that each R_i has the following fields:

(1) time, the index (time) when R_i becomes active.
(2) left, a pointer to the left child of R_i.
(3) right, a pointer to the right child of R_i.
(4) val, the value stored at R_i.
(5) Labels, a list of $O(1)$ labels, each of which is the result of an associative function applied to the values stored at the children of R_i.

Intuitively, each R_i represents a node in a complete binary tree rooted at v whose leaves (which are the same as those of the subtree rooted at v) and are either active or **nil**. The record R_i is active at time R_i.time and remains active until there is a change in one of the descendent nodes of R_i, namely, at time R_{i+1}.time.

More formally, suppose v is a leaf node, which, say, is associated with the element $a \in A$. Also suppose $\sigma(a) = (\sigma_{t_1}, \sigma_{t_2}, \cdots, \sigma_{t_{l_v}})$. Then $B(v) = (R_0, R_1, \cdots, R_{l_v})$, where the record R_i is associated with the operation σ_{t_i}, for $i = 1, 2, \cdots, l_v$. Specifically, given σ_{t_i} in $\sigma(a)$, we define the record R_i so that R_i.time $= t_i$, and R_i.left $= R_i$.right $=$ **nil**. If $\sigma_t = \textbf{insert}(a)$, then R_i.val $= a$, and if $\sigma_t = \textbf{delete}(a)$, then R_i.val $=$ **nil**. Each label in the Labels list is initialized based on R_i.val and the semantics of the function that defines that label. For example, if the label is "number of active descendents," then this label is "1" if R_i.val $= a$, and this label is 0 if R_i.val $=$ **nil**. The record R_0 represents the initial condition, i.e., $R_0 = (0, \textbf{nil}, \textbf{nil}, \textbf{nil}, L_0)$. Intuitively, each record in $B(v)$ is the node in a one-node binary tree that stores a "snapshot" of the history of a with respect to an evaluation of σ.

Now suppose v is an internal node with left child u and right child w. In this case there is a record in $B(v)$ for each record in $B(u) \cup B(w)$. More formally, let $B(u) = (U_0, U_1, \cdots, U_{l_u})$, and $B(w) = (W_0, W_1, \cdots, W_{l_w})$. Also let $(t_0, t_1, t_2, \cdots, t_{l_v})$ be the sorted list of *time* fields from the records in $B(u) \cup B(w)$, where $l_v = l_u + l_w - 1$ (we only store one copy of $t_0 = 0$). We define $B(v) = (R_0, R_1, \cdots, R_{l_v})$, where the record R_i is defined so that R_i.time $= t_i$, R_i.left points to the record U_j with largest index j such that U_j.time $\leq t_i$, and R_i.right points to the record W_j with largest index j such that U_j.time $\leq t_i$. In addition, R_i.val $=$ **nil**, and each label in R_i.Labels is defined by applying the appropriate function to the corresponding labels stored at the records that R_i.left and R_i.right point to. For example, if the label is "number of active descendents," then we simply need to add the corresponding labels from the records R_i.left and R_i.right. Intuitively, each record in $B(v)$ is the root of a binary tree that stores a "snapshot" of the history of the elements associated with the descendents of v with respect to an evaluation of σ.

Atallah, Goodrich, and Kosaraju [4] show that we can exploit the recursive structure of the $B(v)$ definitions to construct $B(v)$ for each v in T in $O(\log n)$ time with $O(n)$ processors in the CREW PRAM, using the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [3].

7.2. The compressed array-of-trees. In our algorithm we use a compressed version of the array-of-trees data structure. The compressed array-of-trees consists of T as above, with a list $B'(v)$ of records stored at each node v in T. The main idea of the compressed array-of-trees is to force each "tree" in $B(\text{root}(T))$ to (1) only store pointers leading to active elements, and (2) not have any internal nodes that have only one child. (See Fig. 5.)

Our method for enforcing this property is as follows. If v is a leaf of T, then the fields of each record in $B'(v)$ are defined as above, i.e., $B'(v) = B(v)$ in this case. If, on the other hand, v is an internal node in T (with left child u and right child w), then we define the structure of the records in $B'(v)$ to be slightly different from the structure of records in $B(v)$. For each record R_i in $B(v)$ there is a record R'_i in $B'(v)$, with R'_i.time $= R_i$.time and R'_i.Labels $= R_i$.Labels. The other fields in R'_i differ from their corresponding fields in R_i, however. In particular, let U denote R_i.left and W denote R_i.right, and let U' and W' denote the records corresponding to U and W in $B'(u)$ and $B'(w)$, respectively. Also let Desc(R) denote the set of all nonnull *val* fields in records reachable from R (by following left and right pointers). We define

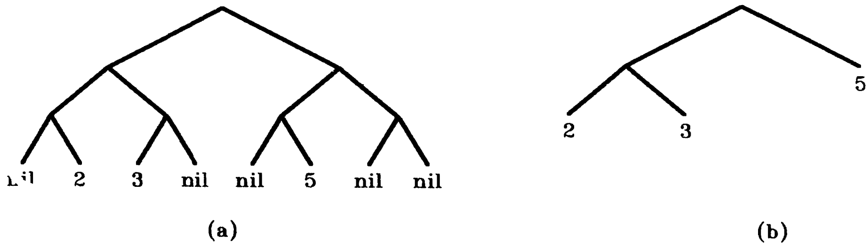


FIG. 5. *An example A_t represented* (a) *as a complete binary tree, and* (b) *as a compressed binary tree.*

the remaining fields of R_i' as follows:

- (1) if $\mathrm{Desc}(R_i) = \emptyset$, then R_i'.left $= R_i'$.right $=$ **nil** and R_i'.val $=$ **nil**.
- (2) If $\mathrm{Desc}(R_i) = \{a\}$, then R_i'.left $= R_i'$.right $=$ **nil** and R_i'.val $= a$.
- (3) If $\mathrm{Desc}(R_i) \neq \emptyset$ but $\mathrm{Desc}(U) = \emptyset$, then R_i'.left $= W'$.left, R_i'.right $= W'$.right, and R_i'.val $= W'$.val.
- (4) If $\mathrm{Desc}(R_i) \neq \emptyset$ but $\mathrm{Desc}(W) = \emptyset$, then R_i'.left $= U'$.left, R_i'.right $= U'$.right, and R_i'.val $= U'$.val.

Note that to construct an R_i' we only need $\mathrm{Desc}(R_i)$ if it contains a single element; otherwise, we need to know only the size of $\mathrm{Desc}(R_i)$. This is itself an associative function. Thus, we can still use the method of Atallah, Goodrich, and Kosaraju [4] to construct $B'(v)$ for each v in T in $O(\log n)$ time with $O(n)$ processors in the CREW PRAM.

7.3. Determining iso-oriented intersections. Having described the compressed array-of-trees, let us return to the problem at hand, namely, the iso-oriented segment arrangement problem. Suppose we are given a set S of n iso-oriented line segments in the plane. We construct the compressed array-of-trees data structure to represent a horizontal plane-sweep (e.g., that of Bentley and Ottmann [6]) and use it to perform a range query for every position i that corresponds to a vertical segment. In particular, we use this data structure by sorting the endpoints of the horizontal segments in S in increasing order by x-coordinates; let Events denote this list. For each point $q_t = (x_t, y_t)$ in Events that is the left endpoint of a segment, we let $\sigma_t = \mathbf{insert}(y_t)$, and for each $q_t = (x_t, y_t)$ in Events that is the right endpoint of a segment, we let $\sigma_t = \mathbf{delete}(y_t)$. The labels we store in the Labels field for each record R in the compressed array-of-trees are $y$max, the maximum y-coordinate in the descendents of R, and desc, the number of active descendents of R. To perform the query for a vertical segment $s = \langle (x, y_1),\ (x, y_2) \rangle$ we first locate the point $q_t = (x_t, y_t)$ in Events such that t is the largest index satisfying $x_t \leq x$ (by a simple binary search). This immediately gives us σ_t, the operation associated with q_t. Intuitively, σ_t is the insertion or deletion event that would be encountered just before the query event for s in a sequential implementation of the plane-sweep. Given σ_t, we locate the record R in $B(\mathrm{root}(T))$ with R.time $= t$. We then perform a search in the tree rooted at R to determine the number k_s of horizontal segments that have a y-value between y_1 and y_2 (using the $y$max and desc labels). This is easily done in $O(\log n)$ time using a single processor. We then assign $\lceil k_s / \log n \rceil$ processors to the task of enumerating these elements and placing them in a single array H_s. The ith processor in this collection is assigned to the task of enumerating the elements in the tree rooted at R

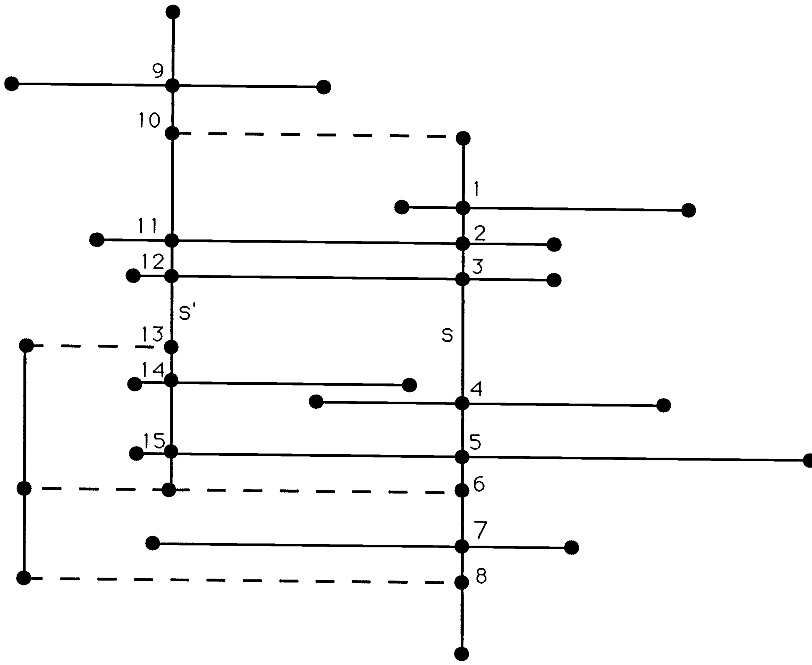


FIG. 6. *Using list merging to complete the construction of the iso-oriented segment arrangement. In this case,* $LV_s = (6,8)$, $RV_{s'} = (10)$, $H_s = (1,2,3,4,5,7)$, $H_{s'} = (9,11,12,14,15)$, $HL_{s',s} = (1,2,3,4,5)$, *and* $HR_{s,s'} = (11,12,14,15)$.

that are in the interval $[y_1, y_2]$ and of rank $i\lceil\log n\rceil, i\lceil\log n\rceil+1, \cdots, (i+1)\lceil\log n\rceil-1$. Since the tree rooted at R is compressed and the elements in its "leaves" are sorted by y-coordinate, the ith processor can use the $y$max and desc labels to locate all its elements in $O(\log n)$ time. Moreover, this also gives us all the vertical adjacencies in the segment arrangement for these intersection points. Thus, we have yet only to combine all the H_s lists to construct the segment arrangement.

We begin this combining procedure by using the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [3] to determine the horizontal shadows of each vertical segment s in S, i.e., the point on the first vertical segment intersected by a horizontal ray emanating out of the endpoints of s. This takes $O(\log n)$ time using $O(n)$ processors [3], and gives us $O(n)$ pairs of segments (s, s') such that s is horizontally "visible" from s'. Then, using parallel sorting [15], in $O(\log n)$ time we can construct, for each vertical segment s, two additional lists: LV_s, which is the sorted list all the horizontal shadows hitting s from the left, and RV_s, which is the sorted list of all the horizontal shadows hitting s from the right. These lists give us all the maximal pieces of s that are visible from another vertical segment in S from either the left or the right.

The remainder of the computation, which we illustrate in Fig. 6, consists of a number of list merging steps, where all lists are assumed to be sorted by y-coordinates. For each s in parallel we merge LV_s with H_s, the list of horizontal intersections along

s. We similarly merge RV_s with H_s. This can be implemented in $O(\log n)$ time using $O(n + k/\log n)$ processors using the merging methods of [9], and [38]. Let $HL_{s,s'}$ be the list of horizontal intersections in H_s that fall on the piece of s that is horizontally visible from s', where s' is to the left of s. Similarly, define $HR_{s,s'}$. Note that we can easily determine each $HL_{s,s'}$ and $HR_{s,s'}$ given the merges we have just performed (even if some of these lists are empty, since we have at least $O(n)$ processors). In parallel, for each pair of horizontally visible segments (s, s') such that s is to left of s', we merge $HL_{s',s}$ with $HR_{s,s'}$. Performing all these parallel merges gives us the horizontal adjacencies for each intersection point in H_s; hence, completes the construction. Since all these merges can also be performed in $O(\log n)$ time using $O(n + k/\log n)$ processors [9], [38], we have the following theorem.

THEOREM 7.1. *Given a set S of n iso-oriented segments in the plane, we can construct the segment arrangement for S in $O(\log n)$ time using $O(n + k/\log n)$ processors in the* CREW PRAM *model, where k is the size of the output.*

8. Conclusion. We have derived a parallel method for constructing the segment arrangement of a set of line segments in the plane in $O(\log n)$ time so that total work performed is only a $\log n$ factor from the sequential lower bound (which is achievable [12]). Moreover, we have shown how to solve the important iso-oriented special case of this problem with an optimal work bound. Thus, the obvious open problem that remains is to construct the segment arrangement in $O(\log n)$ time using only $O(n \log n + k)$ work.

Acknowledgments. We thank Mikhail J. Atallah, Richard Cole, Gregory Bachelis, and S. Rao Kosaraju for helpful discussions regarding the topics of this paper. We also thank an anonymous referee for several helpful comments, which significantly improved the presentation of §6.

REFERENCES

[1] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. Ó'DÚNLAING, AND C. YAP, *Parallel computational geometry*, Algorithmica, 3 (1988), pp. 293–328.

[2] R. ANDERSON, P. BEAME, AND E. BRISSON, *Parallel algorithms for arrangements*, in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece, 1990, pp. 298–306.

[3] M.J. ATALLAH, R. COLE, AND M.T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM J. Comput., 18 (1989), pp. 499–532.

[4] M.J. ATALLAH, M.T. GOODRICH, AND S.R. KOSARAJU, *Parallel algorithms for evaluating sequences of set-manipulation operations*, in Proc. 3rd Aegean Workshop on Computing, AWOC 88, Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 1–10.

[5] B.G. BAUMGART, *A polyhedron representation for computer vision*, Proc. 1975 AFIPS National Computer Conference, 44, AFIPS Press, 1975, pp. 589–596.

[6] J.L. BENTLEY AND T. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., 28 (1979), pp. 643–647.

[7] J.L. BENTLEY AND D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., 29 (1980), pp. 571–576.

[8] S. BHATT AND J.Y. CAI, *Take a Walk, Grow a Tree*, Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, White Plains, NY, IEEE Computer Society, Washington, DC, 1988, pp. 469–478.

[9] G. BILARDI AND A. NICOLAU, *Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines*, SIAM J. Comput., 18 (1989), pp. 216–228.

[10] R.P. BRENT, *The parallel evalutation of general arithmetic expressions,* J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[11] B. CHAZELLE, *Reporting and counting segment intersections,* J. Comput. Systems Sci., 32 (1986), pp. 156–182.

[12] B. CHAZELLE AND H. EDELSBRUNNER, *An optimal algorithm for intersecting line segments in the plane,* Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, White Plains, New York, IEEE Computer Society, Washington, DC, 1988, pp. 590–600.

[13] B. CHAZELLE, L.J. GUIBAS, AND D.T. LEE, *The power of geometric duality,* BIT, 25 (1985), pp. 76–90.

[14] A. CHOW, *Parallel algorithms for geometric problems,* Ph.D. thesis, Computer Science Department, University of Illinois, Urbana, IL, 1980.

[15] R. COLE, *Parallel merge sort,* SIAM J. Comput., 17 (1988), pp. 770–785.

[16] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry,* Springer-Verlag, New York, 1987.

[17] H. EDELSBRUNNER AND L.J. GUIBAS, *Topologically sweeping an arrangement,* in Proc. 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA, Association for Computing Machinery, New York, 1986, pp. 389–403.

[18] H. EDELSBRUNNER, L.J. GUIBAS, J. PACH, R. POLLACK, R. SEIDEL, AND M. SHARIR, *Arrangements of curves in the plane—topology, combinatorics, and algorithms,* UIUCDCS-R-88-1477, Department of Computer Science, University of Illinois, Urbana, IL, 1988.

[19] H. EDELSBRUNNER, J. O'ROURKE, AND R. SEIDEL, *Constructing arrangements of lines and hyperplanes with applications,* in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, AZ, IEEE Computer Society, Washington, DC, 1983, pp. 83–91.

[20] S.K. GHOSH AND D.M. MOUNT, *An output sensitive algorithm for computing visibility graphs,* in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, IEEE Computer Society, Washington, DC, 1987, pp. 11–19.

[21] M.T. GOODRICH, *A polygonal approach to hidden-line elimination,* in Proc. 25th Annual Allerton Conference on Communication, Control, and Computing, Allterton, IL, 1987, pp. 849–858.

[22] M.T. GOODRICH, *Intersecting line segments in parallel with an output-sensitive number of processors,* in Proc. 1989 Annual ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, Association for Computing Machinery, New York, pp. 127–137.

[23] ———, *Constructing arrangements optimally in parallel,* Tech. Report 90/06, Department of Computer Science, The Johns Hopkins University, Baltimore, MD, 1990.

[24] L.J. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams,* ACM Trans. Graphics, 4 (1985), pp. 75–123.

[25] R.H. GÜTING, *An optimal contour algorithm for iso-oriented rectangles,* J. Algorithms, 5 (1984), pp. 303–326.

[26] T. HAGERUP, H. JUNG, AND E. WELZL, *Efficient parallel computation of arrangements of hyperplanes in d dimensions,* in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece, Association for Computing Machinery, New York, 1990, pp. 290–297.

[27] J. HERSHBERGER, *Finding the visibility graph of a simple polygon in time proportional to its size,* in 3rd ACM Symposium on Computational Geometry, Waterloo, Ontario, Canada, Association for Computing Machinery, New York, 1987, pp. 11–20.

[28] C.P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The power of parallel prefix,* in Proc. 1985 Internat. Conference on Parallel Processing, St. Charles, IL, pp. 180–185.

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

[30] D.T. LEE AND F.P. PREPARATA, *Computational geometry—a survey,* IEEE Trans. Comput., 33 (1984), pp. 872–1101.

[31] W. LIPSKI, JR. AND F.P. PREPARATA, *Finding the contour of a union of iso-oriented rectangles,* J. Algorithms, 1 (1980), pp. 235–246.

[32] D.E. MULLER AND F.P. PREPARATA, *Finding the intersection of two convex polyhedra,* Theoret. Comput. Sci., 7 (1978), pp. 217–236.

[33] O. NURMI, *A fast line-sweep algorithm for hidden line elimination,* BIT, 25 (1985), pp. 466–472.

[34] F.P. PREPARATA AND M.I. SHAMOS, *Computational Geometry: An Introduction,* Springer-Verlag, New York, 1985.

[35] J. REIF AND S. SEN, *An efficient output-sensitive hidden-surface removal algorithm and its parallelization,* in Proc. 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, Association for Computing Machinery, New York, 1988, pp. 193–200.

[36] C. RÜB, *Parallele Algorithmen zum Berechnen der Schnittpunkte von Liniensegmenten,* Ph.D. dissertation, Universität des Saarlandes, Saarbrucken, Germany, 1990.

[37] A. SCHMITT, *Time and space bounds for hidden line and hidden surface algorithms,* in Proc. EUROGRAPHICS '81, North-Holland, Amsterdam, 1981, pp. 43–56.

[38] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model,* J. Algorithms, 2 (1981), pp. 88–102.

[39] D. WOOD, *The contour problem for rectilinear polygons,* Inform. Process. Lett. 19 (1984), pp. 229–236.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 756–765, August 1991
© 1991 Society for Industrial and Applied Mathematics
011

FAST PARALLEL ARITHMETIC VIA MODULAR REPRESENTATION*

GEORGE I. DAVIDA† AND BRUCE LITOW†

Abstract. An almost uniform NC^1 circuit family for integer division is presented. The circuit size is $O(n^6/\log(n))$. The circuit design is based on modular representation for integers below 2^n. In particular, a very efficient technique is introduced for computing "$a < b$?" when a and b are in modular representation. This leads to a uniform NC^1 circuit of $O(n^2)$ size for comparison of integers in modular representation.

Key words. uniform, almost uniform, parallelism, circuit, combinational, modular, comparison, division

AMS(MOS) subject classifications. 68Q10, 68Q25

1. Introduction. In this paper we work within the NC^1 computation model which consists of families of Boolean circuits of $O(\log(n))$ depth and $n^{O(1)}$ size. Actually the definition of NC^1 entails uniformity issues. For a discussion of uniformity notions and their relations to other complexity issues, see [6], [2], and [3]. It is known that all four arithmetic operations: $+$, $-$, $/$, $\times$ can be computed by NC^1 circuit families. A discussion of these circuits is contained in [11]. The paper in which division was first accomplished in $O(\log(n))$ depth is [1]. This circuit family is P-uniform and has size $O(n^4 \log^3(n))$. The families for $+$, $-$, and $\times$ are LOGSPACE uniform. However, the circuit family for division in [1] is not known to be LOGSPACE uniform. We will henceforth refer to LOGSPACE uniform as simply uniform. By almost uniform we mean DSPACE$(\log(n) \log\log(n))$ or smaller. In this paper we show that division can be computed by an almost uniform NC^1 family of $O(n^6/\log(n))$ size. Almost uniformity is achieved using a method in [5].

Motivation for our circuit design comes from more traditional approaches to the parallelization of arithmetic operations. In §2 we develop a new approach to modular (sometimes called residue) representation of integers in order to compute the standard order $<$ efficiently. In §3 we use modular representation in a division algorithm and show in §4 that this algorithm can be implemented by an almost uniform NC^1 family.

2. Modular representation. We briefly recapitulate a few basic facts about modular representation. We will call it Chinese remainder representation or CRR. For more details consult [4], [8]. Throughout the paper we let $m_1, \cdots, m_r$ be odd, pairwise relatively prime positive integers. We define the CRR product modulus to be $M = \prod_{j=1}^{r} m_j$. Let x be an integer; then we associate with it the vector $\vec{x} = (x_1, \cdots, x_r)$ where $0 \leq x_j < m_j$ and $x = x_j \bmod m_j$. Following [8] we will write $|x|_a$ for $x \bmod a$. We will work with nonnegative integers and always take $0 \leq |x|_a < a$. Next put $M_j = M/m_j$ and $\hat{M}_j = |M_j^{m_j - 1}|_M$. Note that

$$|\hat{M}_j|_{m_k} = \begin{cases} 1 & \text{if j = k,} \\ 0 & \text{if j} \neq \text{k,} \end{cases}$$

which is central in proving the following theorem (see [4]).

THEOREM 2.1 (Chinese remainder theorem). $|x|_M = |\sum_{j=1}^{r} x_j \hat{M}_j|_M$.

* Received by the editors February 9, 1989, accepted for publication (in revised form) August 16, 1990.

† Computer Science Department, University of Wisconsin, Milwaukee, Wisconsin 53201.

Note that Theorem 2.1 implies that if $0 \leq x, y < M$, then $\vec{x} = \vec{y}$ if and only if $x = y$. Let $\circ$ be either $+$, $-$, or $\times$ and let $z = x \circ y$. As a corollary to the Chinese remainder theorem we have

$$\vec{z} = (|x_1 \circ y_1|_{m_1}, \cdots, |x_r \circ y_r|_{m_r}),$$

which provides an obvious starting point for parallelization. However, it is not clear how to compute the standard order $<$, and so overflow detection in the case of $+$, $\times$, and detection of negative integers in the case of $-$, are problems. It is even less clear how to perform division in modular representation in an efficient manner. Some effort to confront these difficulties has been made. An interesting approach to handling $<$ is given in [10]. We give another approach to this problem.

Before commencing with the technical exposition we want to emphasize to the reader that unless explicitly indicated as a binary representation, all integers, e.g., x, y, etc., are represented as CRR vectors. That is, all arithmetic operations and comparison involving x and y, etc., are carried out in CRR.

We refer to Theorem 2.1 as CRT. CRT in its present form is not quite what we need. Put $c_j = |x_j M_j^{m_j - 2}|_{m_j}$. The following equation is easily verified:

$$\left| \sum_{j=1}^{r} c_j M_j \right|_{m_k} = \left| \sum_{j=1}^{r} x_j |\hat{M}_j|_{m_j} \right|_{m_k} = x_k.$$

If $0 \leq x < M$, then CRT tells us that

$$\sum_{j=1}^{r} c_j M_j = J(x)M + x, \tag{1}$$

where $J(x)$ is an integer, $0 \leq J(x) < r$. Dividing through (1) by M, we get the starting point for our treatment of CRR:

$$\sum_{j=1}^{r} c_j / m_j = J(x) + x/M. \tag{2}$$

We now develop a way to compute $<$ for CRR. Let $0 \leq x < M$. All CRR-related quantities are tied to x. Let q be the least positive integer such that

$$r \cdot 2^{-q} < \frac{1}{4} \tag{3}$$

and let σ_j be the first q bits in the binary expansion of c_j/m_j approximating it from below. Note that $q = O(\log r)$. From (3) and the triangle inequality we get

$$0 \leq \sum_{j=1}^{r} \left(\frac{c_j}{m_j} - \sigma_j \right) \leq r \cdot 2^{-q} < \frac{1}{4}. \tag{4}$$

If α is real and $n \leq \alpha < n + 1$ where n is an integer, then put $\lfloor \alpha \rfloor = n$ and put $\langle \alpha \rangle = \alpha - \lfloor \alpha \rfloor$. We abbreviate $\sum_{j=1}^{r} \sigma_j$ by σ. When we want to indicate explicitly the association with x we write $\sigma(x)$.

Our principal objective is to compute $J(x)$. To do this we will work with (1), mod 2. First we collect some basic facts.

LEMMA 2.2. *If* $\langle\sigma\rangle \le \frac{3}{4}$, *then* $\lfloor\sigma\rfloor = J(x)$.

Proof. We can write $\sum_{j=1}^{r} c_j/m_j = \lfloor\sigma\rfloor + \langle\sigma\rangle + \alpha$, where $0 \le \alpha < \frac{1}{4}$ by (4). The result now follows from (2). □

LEMMA 2.3. *If* $\frac{1}{4} \le x/M \le \frac{3}{4}$, *then* $\langle\sigma\rangle \le \frac{3}{4}$.

Proof. Following the proof of Lemma 2.2 we have

$$\langle\sigma\rangle = J(x) - \lfloor\sigma\rfloor + x/M - \alpha$$

but now $0 < x/M - \alpha \le \frac{3}{4}$. This implies that $J(x) = \lfloor\sigma\rfloor$ and also the lemma. □

We would like to compute $J(x)$ since it plays a key role in subsequent computations. By Lemmas 2.2 and 2.3 we see that if $\frac{1}{4} \le x/M < 1$ we do have $\lfloor\sigma\rfloor = J(x)$. However, if $x/M < \frac{1}{4}$, then $\lfloor\sigma\rfloor$ may equal $J(x) - 1$. We give an example. Let m_1, m_2, m_3 be $3, 5, 7$ so $M = 105$. Pick $x = 2$; then we get $(c_1, c_2, c_3) = (1, 2, 2)$. From (3) we pick $q = 4$. This gives $\sigma_1 = .0101, \sigma_2 = .0110, \sigma_3 = .0100$. Thus we get $\sigma = .1111$, so $\lfloor\sigma\rfloor = 0$. Evaluating (1) we see that $J(2) = 1$. We next develop a way to obtain $J(x)$ in any case.

We define a tree-type Boolean circuit T_n. In order to simplify the discussion we will allow each node to process two inputs, each of which is in $\{0,1\}^2$ rather than $\{0,1\}$. An interior node with left input $a = (a_1, a_2)$ and right input $b = (b_1, b_2)$ will be assigned:

$$\begin{array}{ll} (1, a_2) & \text{if } a_1 = 1, \\ (1, b_2) & \text{if } a_1 = 0 \text{ and } b_1 = 1, \\ (0, 0) & \text{otherwise.} \end{array}$$

T_n is a binary tree built bottom-up by pairing leaves starting at the left. Leaves are numbered $0, 1, \cdots, n-1$, consecutively from the left. The following lemma is immediate.

LEMMA 2.4. *If j is least such that leaf j receives an input with the left component equal to* 1, *then the root will receive this value within* $\log(n)$ *steps. If no such leaf exists, then the root receives* $(0,0)$ *in* $\log(n)$ *steps.*

We now apply T_n to computing $J(x)$. Initialize the leaves as follows.

$$\mathrm{LEAF}_k = \begin{cases} (0,0) & \text{if } \langle\sigma(2^k x)\rangle > \frac{3}{4}, \\ (1, ||2^k x|_M|_2) & \text{if } \langle\sigma(2^k x)\rangle \le \frac{3}{4}. \end{cases}$$

LEMMA 2.5. *If* $\langle\sigma\rangle > \frac{3}{4}$, *then the root of* T_n *gets the value* $(1,0)$ *if and only if* $2x < M$.

Proof. ($\Rightarrow$) Let us first examine what happens when $x/M < \frac{1}{4}$. That is, for some $2 < h < n, M/2^{h+1} < x < M/2^h$. Thus for $k = h - 1$ we have $\frac{1}{4} < 2^k x/M < \frac{1}{2}$. Thus Lemma 2.2 implies that $\langle\sigma(2^k x)\rangle \le \frac{3}{4}$. This means that LEAF_k gets the input $(1, |2^k x|_M|_2)$ but since $|2^k x|_M = 2^k x$ in this case we see that LEAF_k gets $(1,0)$. It may happen that for some $j < k$, $\langle\sigma(2^j x)\rangle \le \frac{3}{4}$, but again LEAF_j gets $(1,0)$. Thus by Lemma 2.3, the root gets the value $(1,0)$.

Next let us look at $x/M > \frac{3}{4}$. We can write $x = M - y$, where $y/M < \frac{1}{4}$. Note that $|2^k x|_M = |M - 2^k y|_M$. By the previous paragraph there must be a k such that $\frac{1}{4} < 2^k y/M < \frac{1}{2}$. Thus $|M - 2^k y|_M = |2^k x|_M = M - 2^k y$ and since $\frac{1}{2} < |2^k x|_M/M < \frac{3}{4}$, Lemma 2.3 implies that $\langle\sigma(2^k x)\rangle \le \frac{3}{4}$. Therefore LEAF_k gets $(1,1)$ since $|M - 2^k y|_2 = 1$. Of course it may happen that for some $j < k$, $\langle\sigma(2^j x)\rangle \le \frac{3}{4}$ but again we get $|M - 2^k y|_2 = 1$.

($\Leftarrow$) Now if $2x < M$, then by Lemma 2.3, $x/M < \frac{1}{4}$, so that by the first paragraph of the proof the root gets $(1,0)$. If $2x > M$, then Lemma 2.3 implies that $\frac{3}{4} < x/M$ and by the second paragraph the root gets $(1,1)$. □

We give a method for deciding when $2x < M$. We call this method Φ_0. Let H be the least even positive integer such that $2H > M$. Put $U = (M-1)/2$ and assume that U is odd. This assumption is not really necessary but it simplifies the exposition.

ALGORITHM Φ_0.
Φ_0.1 Compute σ.
Φ_0.2 If $\langle\sigma\rangle > \frac{3}{4}$, GOTO 11
Φ_0.3 Compute $|x|_2 = |\sum_{j=1}^{r} c_j M_j + \lfloor\sigma\rfloor \cdot M|_2$
Φ_0.4 $y \leftarrow |x+H|_M$
Φ_0.5 Compute $\sigma(y)$
Φ_0.6 If $\langle\sigma(y)\rangle > \frac{3}{4}$ GOTO 9
Φ_0.7 Compute $|y|_2 = |\sum_{j=1}^{r} c_j(y) M_j + \lfloor\sigma(y)\rfloor \cdot M|_2$.
Φ_0.8 If $|x|_2 = |y|_2$, then EXIT$(2x < M)$, ELSE EXIT$(2x > M)$
Φ_0.9 Compute $T_n(y)$
Φ_0.10 If the root gets $(1,1)$, then EXIT$(2x < M)$, ELSE EXIT $(2x > M)$
Φ_0.11 Compute $T_n(x)$
Φ_0.12 If the root gets $(1,0)$, then EXIT$(2x < M)$, ELSE EXIT$(2x > M)$

LEMMA 2.6. *If $x \neq U$, then Φ_0 is correct.*

Proof. First we make the observation that $2x < M$ if and only if $x + H < M$, assuming that $x \neq U$. Steps Φ_0.3 and Φ_0.7 are correct by Lemma 2.2 and (1). Step Φ_0.8 is correct since at that point $y = x+H$; otherwise $|y|_2 \neq |x+H|_2$. Step Φ_0.10 is correct since at that point we can use Lemma 2.3. Step Φ_0.12 is correct since again we can use Lemma 2.5. □

Remark. Note that $\vec{U}$ is easy to compute. Furthermore, we can test if $\vec{x} = \vec{U}$ in $O(\log(n))$. Also note that if U is even, then we modify Φ_0. Now we exclude both U and $U-1$. Again we can easily test for $U-1$ in $O(\log(n))$.

It remains to compute $J(x)$. By Lemma 2.2, if $\langle\sigma\rangle \leq \frac{3}{4}$, then we are done. Otherwise we use Φ_0 to decide whether $2x < M$.

LEMMA 2.7. *If $2x > M$ and $\langle\sigma\rangle > \frac{3}{4}$, then $J(x) = \lfloor\sigma\rfloor$.*

Proof. By Lemma 2.3 and hypothesis, $x/M > \frac{3}{4}$. From the proof of Lemma 2.3, we have the result. □

We can now compute $J(x)$. The problem case is where $2x < M$ and $\langle\sigma\rangle > \frac{3}{4}$. Put $y = M - x$ so that $2y > M$. Now in any case we can compute $J(y)$. Thus taking (1) mod 2 we can obtain $|y|_2$. Next use (1) mod 2 to attempt to find $|x|_2$. Call this value c. Here we use $\lfloor\sigma\rfloor$ in place of $J(x)$ which is still unknown. Now we know that $|x|_2 = ||y_2| + 1|_2$. If $c = ||y|_2 + 1|_2$, then $J(x) = \lfloor\sigma\rfloor$; otherwise $J(x) = \lfloor\sigma\rfloor + 1$. This conclusion follows from the proof of Lemma 2.3. We employ these observations in the following algorithm.

Finally we describe an algorithm for computing $<$. The inputs are two n bit integers, x and y.

ALGORITHM Φ_1.
Φ_1.1 Use Algorithm Φ_0. If $2x < M$ and $2y > M$, then EXIT$(x < y)$
Φ_1.2 Use Algorithm Φ_0. If $2y < M$ and $2x > M$, then EXIT$(y < x)$
Φ_1.3 Put $z = |x - y|_M$
Φ_1.4 Compute $|x|_2, |y|_2, |z|_2$
Φ_1.5 If $|z|_2 = ||x|_2 + |y|_2|_2$, then EXIT$(y < x)$

Φ_1.6 EXIT$(x < y)$

LEMMA 2.8. *Φ_1 is correct.*

Proof. If $y < x$, then $z = x - y$; otherwise $z = M + x - y$. □

3. Division. Let $y < x < 2^n < M$ where $2^{k-1} \le y < 2^k$. We want to compute $\lfloor x/y \rfloor$. Define the nonnegative integer h by $2^{n-1} \le 2^h y < 2^n$. Since $2^h x / 2^h y = x/y$ we will work with $x_h \leftarrow 2^h x$ and $y_h \leftarrow 2^h y$. Now we have $2^{n-1} \le y_h < 2^n$ and $2^{n-1} < x_h < 2^{2n}$. Let z be the integer, $0 \le z < 2^{n-1}$, such that $y_h = 2^n - z$ so that setting $\beta = z/2^n$, $\beta < \frac{1}{2}$. We have

$$x_h/y_h = x_h \cdot 2^{-n} \cdot \sum_{j=0}^{\infty} \beta^j$$

by geometric series summation for $(1-\beta)^{-1}$. Put

$$A = x_h 2^{-n} \sum_{j=0}^{n+1} \beta^j, \qquad B = x_h 2^{-n} \sum_{j=n+2}^{\infty} \beta^j$$

and note that $B < x_h 2^{-n} \beta^{n+1}$ since $\beta < \frac{1}{2}$. Thus $B < \frac{1}{2}$. Put

$$C = x_h \sum_{j=0}^{n+1} z^j 2^{n(n+1-j)} \tag{5}$$

and observe that $A = 2^{-n(n+2)} C$. Now since $z < 2^{n-1}$,

$$C < 2^{2n} \sum_{j=0}^{n+1} 2^{nj-j+n(n+1)-nj} = 2^{2n+n(n+1)} \sum_{j=0}^{n+1} 2^{-j} < 2^{n(n+3)}.$$

These ideas lead to algorithm Φ_2.

ALGORITHM Φ_2. Compute $\lfloor x/y \rfloor$.

Φ_2.1 Compute h such that $2^{n-1} \le 2^h y < 2^n$

Φ_2.2 $x_h \leftarrow 2^h x, y_h \leftarrow 2^h y, z \leftarrow 2^n - y_h$

Φ_2.3 Compute $D = \lfloor A \rfloor$

Φ_2.4 If $x_h - D y_h < y_h$, then EXIT(D), ELSE EXIT$(D+1)$

From the above discussion it is clear that Φ_2 is correct.

It remains to discuss the computation of step 3 of Φ_2. The algorithm for step 3 of Φ_2 is called Φ_3. Here we will explicitly use the integer C, introduced in (5). We let $M' > 2^{n(n+3)}$ be the product modulus of a CRR. All primed symbols refer to that CRR.

ALGORITHM Φ_3. Compute step 3 of Φ_2.

Φ_3.1 $E = |C|_{2^{n(n+2)}} = | \sum_{j=1}^{r} c'_j(C) M'_j - J'(C) M' |_{2^{n(n+2)}}$

Φ_3.2 Compute u such that $|u \cdot 2^{n(n+2)}|_{M'} = 1$

Φ_3.3 $\lfloor (C - E) \cdot 2^{-n(n+2)} \rfloor = |(C - E) \cdot u|_{M'} = D$

LEMMA 3.1. *Φ_3 is correct.*

Proof. Step Φ_3.1 follows from (1). We use Φ_0 to compute $J'(C)$. Step Φ_3.2 is always possible since GCD$(2, M') = 1$. Step Φ_3.3 follows from steps Φ_3.1 and Φ_3.2. □

In this section and in §2 we have not provided any implementation details. In the following section we will discuss how to break down the steps of the Φ algorithms in terms of Boolean computations.

4. Circuits. This section is divided into two parts. In the first part we will determine the sizes and depths of the basic Boolean subcircuits for division. In the second part we consider the uniformity issue. We emphasize that the inputs and outputs of our circuits are binary strings which are assumed to be CRR tuples rather than binary notation.

4.1. Basic circuits. First we make an observation about CRR.

LEMMA 4.1. *If $2^J < M < 2^{J+1}$, then primes $m_1 < \cdots < m_r$ can be uniformly generated so that $m_r < J, r < O(J/\log J)$.*

Proof. It is the case that there are $O(J/\log(J))$ primes between $J/2$ and J (see [7]). Let k be the largest integer such that $2^k < J$. Thus we require that $2^{kr} > 2^J$ or $kr > J$. That is, we have $r \in O(J/\log(J))$ and $m_r < 2^k < J$. □

Refer to the number C, (5), that is used in Φ_3. The magnitude of C forces us to use two CRRs. CRR(n) is the CRR we use for inputs and outputs from the division algorithm. The critical point is that intermediate values will use CRR($2n^2+1$), i.e., a CRR sufficient for all $0 \le x < 2^{2n^2}$. Here we choose n large enough so that $2^{n(n+3)} < 2^{2n^2}$. CRR($2n^2+1$) will use the prime moduli $m_1 < \cdots m_r < m_{r+1} < \cdots m_\rho$. All moduli are chosen in accordance with Lemma 4.1 and $m_1, \cdots, m_r$ are the CRR(n) moduli. We define $M' = \prod_{j=1}^{\rho} m_j$ and $M'_j = M'/m_j$ for $1 \le j \le \rho$. Thus $2^{2n^2} < M'$. All primed symbols will be understood as coming from CRR($2n^2+1$). Note that by Lemma 4.1, we may take $\rho \in O(n^2/\log(n))$.

Every step of the Φ algorithms ultimately involves computations of the following kinds. All the numbers occurring in these computations are $O(\log(n))$ bit binary integers by Lemma 4.1 and (3).

BASIC ARITHMETIC OPERATIONS.
OP.1 Determine whether $a = b$
OP.2 Build binary tree circuits of depth $O(\log(n))$ and size $O(n)$
OP.3 Compute σ_j
OP.4 Compute σ
OP.5 Compute $|a \circ b|_{m_j}$ for $\circ = +, -, \times$
OP.6 Compute $|a^b|_{m_j}$

OP.1 and OP.2 are trivial. We turn next to OP.3 and OP.5.

LEMMA 4.2. OP.3 *and* OP.5 *can be computed by uniform $O(\log(n))$ depth, $O(n \log^{O(1)}(n))$ size circuits.*

Proof. OP.3 amounts to computing $\lfloor a/b \rfloor$ for $O(\log(n))$ bit integers, a, b. Depth and size follow from, e.g., [5], [1]. Bounds for OP.5 follow from the same considerations. □

We define three tables.

$T^{(1)}$ is a one-dimensional table where $T_j^{(1)}$ contains $\vec{z}$, in CRR($2n^2+1$), where $z = |M'_j|_{2^{n(n+2)}}, 1 \le j \le \rho+1$ and we put $M' = M_{\rho+1}$.

$T^{(2)}$ is a two-dimensional table where $T_{j,k}^{(2)}$ contains the binary representation of $M_j \bmod m_k$, where $1 \le j \le r$ and $r+1 \le k \le \rho$.

$T^{(3)}$ is a two-dimensional table where $T_{j,k}^{(3)}$ contains the binary representation of $f_j^k \bmod m_j$, where $1 \le j \le \rho$, $1 \le k < 2n^2$, and f_j is primitive for m_j.

LEMMA 4.3.
- $\text{SIZE}(T^{(1)}) \in O(\rho \cdot n^2) = O(n^4/\log(n))$.
- $\text{SIZE}(T^{(2)}) \in O(r \cdot \rho \cdot \log(n)) = O(n^3)$.
- $\text{SIZE}(T^{(3)}) \in O(\rho \cdot n^2 \cdot \log(n)) = O(n^4)$.

Proof. We use Lemma 4.1 and note that in (3) we can pick $q = \log(r) + 3$. □

LEMMA 4.4. *OP.6 can be computed in $O(\log(n))$ depth and $O(n^4)$ size.*

Proof. Since f_g is primitive mod m_g and $m_g < n$, we know that $a = |f_g^k|_{m_g}$ for $k < n$. Thus we use $T^{(3)}$, indexed first by m_g, then a to find k. Next compute bk in binary. Note that $bk < n^2$, so we can again use $T^{(3)}$ to look up $|f_g^{bk}|_{m_g} = |a^b|_{m_g}$. □

Remark 1. If $b < n$, then Lemma 4.4 can be amended to a size bound of $O(n^3)$ since $T^{(3)}$ need only involve powers of primitive elements no higher than the nth and so has $O(n^3)$ size.

Remark 2. Observe by Lemma 4.4 and Remark 1 following it that for $x < 2^{2n^2}$ and $b < n, |x^b|_{M'}$ requires $O(\log(n))$ depth and $O(n^6/\log(n))$ size since in CRR this reduces to ρ parallel instances of OP.6. It also follows that $|x^{-1}|_{M'}$, when it exists, requires the same depth and size, since this again reduces to ρ parallel instances of OP.6.

The following is probably well known but we give it for completeness. Let $z_1, \cdots, z_n$ be p bit integers and put $z = \sum_{j=1}^{n} z_j$. We assume that $0 < p \leq \log(n)$.

LEMMA 4.5. *There is a Boolean circuit of size $O(n \cdot p)$ and depth $O(\log(n))$ which computes z.*

Proof. Construct p copies of a circuit which can add up to n bits. That is, the value computed by any such circuit will be an integer requiring at most $\log(n)$ bits. The jth such circuit will add together the values of the jth positions in each of the z_k. Clearly [11, p. 75], each such circuit has depth $O(\log(n))$ and size $O(n)$. The result is essentially a list of size p consisting of numbers each needing fewer than $2\log(n)$ bits. We use here the fact that $p \leq \log(n)$ which limits bit size to below $\log(n) + p$. Notice, for example, that the the sum of all n high-order bits will have to be padded with p low-order 0s. This accounts for the summand of p. It is now clear that the sum of this list can be straightforwardly evaluated in $O(\log\log^2(n))$ depth which is dominated by $O(\log(n))$. It is also clear that total circuit size is $O(pn)$. □

Remark. Lemma 4.5 implies that OP.4 requires depth $O(\log(n))$ and size $O(n\log(n))$. We have accounted for all the basic arithmetic operations needed in the Φ algorithms.

We next give sizes and depths for Boolean circuits for each step in Φ_0. All numbers are represented by CRR($2n$). Refer to the comment following Lemma 4.1 on why this does not affect the size and depth bounds.

Φ_0.1 and Φ_0.5. Depth $= O(\log(n))$, SIZE $= O(n\log(n))$ by Lemma 4.5.

Φ_0.2 and Φ_0.6. Note that $\langle\sigma\rangle$ is a $q = O(\log(n))$ bit number. DEPTH $= O(\log\log(n))$, SIZE $= O(\log(n))$.

Φ_0.3 and Φ_0.7. Note that $|M|_2 = 1 = |M_j|_2$ and since σ is a q bit binary number we obtain $||\lfloor\sigma\rfloor|_2$ in $O(1)$ depth and q size. Similarly, the c_j are $O(\log(n))$ bit numbers. Finally, the r-fold summation can be done in $O(\log(n))$ depth. DEPTH $= O(\log(n))$, SIZE $= O(n)$.

Φ_0.4 and Φ_0.8. SIZE $= O(n)$, DEPTH $= O(\log(n))$ by Lemma 4.1.

Φ_0.9–Φ_0.12. Clearly, Φ_0.10 and Φ_0.12 are trivial. Φ_0.9 and Φ_0.11 are the most size intensive. In these steps the initialization of the leaves involves $T^{(3)}$. Since we are working with CRR(n) the size of $T^{(3)}$ drops to $O(r \cdot n \cdot \log(n))$ (see Lemma 4.3), which gives $O(n^2)$ size for that table. Now each entry of $T^{(3)}$ is hard wired to a leaf of the tree. The kth leaf getting the kth entry. Since table look-up is not required, only one copy of $T^{(3)}$ is used. Thus we get $O(n^2)$ size and $O(\log(n))$ depth.

THEOREM 4.6. *Comparison of $O(n)$ bit* CRR *tuples can be done by uniform NC^1 circuits of size $O(n^2)$.*

Proof. The only steps requiring comment are $\Phi_1.1$, $\Phi_1.2$, and $\Phi_1.4$. In these three steps we are using Φ_0. The size and depth claims follow from the above analysis of Φ_0. It is straightforward that $T^{(3)}$ can be produced in LOGSPACE and the uniformity claim follows from that. □

We now give sizes and depths for circuits in Φ_2.

Φ_2.1. This step uses CRR(n). In parallel for each $0 \le h < n$, test, using Φ_1, whether or not $2^{n-1} \le 2^h y$. The size is dominated by n copies of the Φ_1 comparison circuit so that by Theorem 4.1 we get $O(n^3)$ size and $O(\log(n))$ depth.

Φ_2.2. This step uses CRR(n). SIZE $= O(n \log^{O(1)}(n))$ and DEPTH $= O(\log(n))$ by OP.5.

Φ_2.3. This step is complicated, dominates the sizes of all other steps, and uses CRR($2n^2+1$). We treat the computation of C here and defer computation of D to the discussion of Φ_3. In order to compute C we must convert x_h, y_h, z, and 2^n to CRR($2n^2$) tuples. This is done using $T^{(2)}$, much as steps $\Phi_0.3$ and $\Phi_0.7$ were computed, except that now we work mod m_g rather than mod 2. For example, to compute $|x_h|_{m_g}$ for some $r+1 \le g \le \rho$ we use

$$|x_h|_{m_g} = \left| \sum_{j=1}^{r} c_j(x_h) M_j - J(x_h) M \right|_{m_g}.$$

Note that the $c_j(x_h)$ and $J(x_h)$ are computed in CRR(n) and we use OP.5. This gives $O(\log(n))$ depth by Lemma 4.5. Size is dominated by the size of $T^{(2)}$ which is $O(n^3 \log(n))$. Next we require z^j for $1 \le j \le n+1$. To do this we appeal to Remark 2 following Lemma 4.4, which gives $O(n^5/\log(n))$ size and $O(\log(n))$ depth for each z^j; hence an overall size of $O(n^6/\log(n))$. We can hard wire the $2^{n(n+1-j)}$ using the $O(n^4)$ size version of $T^{(3)}$. The summation size is dominated by the CRR($2n^2$) for the $O(n)$ numbers $z^j, 2^{n(n+1-j)}$ which is $O(n^3)$. Thus the total size is $O(n^6/\log(n))$.

Finally we must perform the n-fold addition to get C. This amounts to computing $|\sum_{j=1}^{n+1} z^j 2^{n(n+1-j)}|_{m_g}$ for each $1 \le g \le \rho$. By Lemma 4.5 we get $O(\log(n))$ depth and $O(n \log(n) \rho) = O(n^3)$ size for all these summations. Thus the overall size is dominated by $T^{(2)}$, which is $O(n^3 \log(n))$.

Φ_2.4. D is in CRR($2n^2+1$), however, since it is smaller than 2^n we can work in CRR(n) at this point. Now x_h, y_h are available in CRR(n). It is trivial to convert from CRR($2n^2+1$) to CRR(n)—just drop all CRR vector components beyond the rth. By Theorem 4.1 we get $O(\log(n))$ depth and $O(n^2)$ size.

We now give sizes and depths for circuits of Φ_3. All tuples are in CRR($2n^2$).

Φ_3.1. We need $T^{(1)}$ to evaluate the sum and we need to hard wire one copy of the $O(n^4)$ size $T^{(3)}$ in order to compute $J'(C)$ via Φ_0. Size is dominated by $T^{(3)}$, which is $O(n^4)$ and depth is $O(\log(n))$ by Lemma 4.5.

Φ_3.2. Recall that the computation of C was discussed in step Φ_2.2. The other computations in this step also yield SIZE $= O(n^3)$, DEPTH $= O(\log(n))$ by Lemma 4.4 and the remark following it.

Φ_3.3. SIZE $= O(n^2)$ and DEPTH $= O(\log(n))$.

Summarizing the accounting we have done in this section and noting in particular that step Φ_2.3 dominates the sizes of all other steps, we have Theorem 4.7.

THEOREM 4.7. *Division can be done by P uniform NC^1 circuits of size $O(n^6/\log(n))$.* □

Remark. With the exception of the three tables it is already clear that all other circuits can be produced in LOGSPACE. We discuss uniformity considerations for the tables in the next section.

4.2. Uniformity. The obstacle to uniformity is $T^{(1)}$. It is clear that this table can be produced in P. However, we can say a little more. We assume that n is a power of 2. If not, then we multiply each n bit x by the least power of 2 such that the resulting bit size is a power of 2. This cannot increase bit size by more than a factor of 2. Referring to A and B used with C, (5), to motivate Φ_2 and Φ_3, we let the summation for A go from $j = 0$ to $2n - 1$. Thus B summation now starts at $2n$. This does not materially affect the analysis of the division circuit. In fact the error term, B, is made smaller. Now we get

$$A = 2^{-2n^2} C, \tag{6}$$

i.e., we are now working with a power of 2 whose exponent , $2n^2$, is itself a power of 2. This will make possible an application of a result in [5]. Next we describe a modified $T^{(1)}$, denoted by T. T is indexed just as was $T^{(1)}$, but its entries are CRR($2n^2 + 1$) vectors for the $|M_j'|_{2^{2n^2}+1}$.

LEMMA 4.8. *Assume that an $n^{O(1)}$ length binary representation of an integer, x, is being produced by a transducer in $O(\log(n))$ space. Let m be a $\log(n)$ bit number. Then $|x|_m$ can be produced in $O(\log(n))$ space.*

Proof. Let h be the integer such that $2^{h-1} \leq m < 2^h$. We informally describe a transducer. As the bits of x arrive, we segment them into length h blocks. Each segment is treated as the binary representation of a number below 2^h. Clearly, we can reduce this mod m in LOGSPACE. The segments correspond to the powers $2^0, 2^h, 2^{2h}, \cdots$. It is also clear that, given $|2^{gh}|_m$, we can obtain $|2^{(g+1)h}|_m$ in LOGSPACE. Thus all intermediate results can be written in LOGSPACE ($O(h)$). □

LEMMA 4.9. *T can be produced in* DSPACE($\log(n) \log\log(n)$).

Proof. We illustrate the procedure with the production of the CRR($2n^2 + 1$) vector for $|M'|_{2^{2n^2}+1}$. From [5] there is a uniform $n^{O(1)}$ size, $O(\log(\rho) \log\log(n))$ depth circuit that will produce (in binary)

$$V = |V'|_{2^{2n^2}+1}, \quad \text{where} \quad V' = \prod_{j=1}^{\rho} v_j$$

and where each $v_j = 2^{2n^2} + 1 - m_j$, so that each v_j needs $2n^2$ bits. It is clear that the v_j can be produced in $O(\log(n))$ depth. We will assume without loss of generality that ρ is even. Then we can write $V' = (2^{2n^2} + 1)V'' + M'$. Thus $V = |M'|_{2^{2n^2}+1}$. We now apply Lemma 4.6 to get the CRR($2n^2 + 1 + 1$) vector for V. The lemma follows from Theorem 4 of [2]. □

We show that T will work just as well as $T^{(1)}$. First Lemma 4.1 can be modified so that $\text{GCD}(M', 2^{2n^2} + 1) = 1$. We must simply omit from the m_j all prime divisors of $2^{2n^2} + 1$. This number has fewer than, say, $2\log(n)$ such divisors, so Lemma 4.1 still goes through. Second we must recover D in step 3 of Φ_2 from $\lfloor C/(2^{2n^2} + 1) \rfloor$. Noting (6), we now have $D = \lfloor A \rfloor = \lfloor 2^{-2n^2} C \rfloor$. Assume that $\lfloor C/(2^{2n^2} + 1) \rfloor$ has been computed. Now

$$C/(2^{2n^2} + 1) = C/2^{2n^2} - C/2^{4n^2} + C/2^{6n^2} - \cdots$$

and since $C < 2^{4n^2-1}$ it is clear that

$$0 \leq C/2^{2n^2} - C/(2^{2n^2}+1) < \frac{1}{2}$$

and so we can determine $\lfloor C/2^{2n^2} \rfloor$ from $\lfloor C/(2^{2n^2}+1) \rfloor$ in $O(\log(n))$ depth. Hence we can obtain D. Finally we can precompute $|(2^{2n^2}+1)^{-1}|_{M'}$ in LOGSPACE by combining Remark 2 following Lemma 4.4 and Theorem 4 of [2].

We can now establish the division circuit complexity.

THEOREM 4.10. *Division of $O(n)$ bit integers in* CRR *can be done by almost uniform circuits of $O(\log(n))$ depth and size $O(n^6/\log(n))$.*

Proof. The only remaining issue is uniformity. By Theorem 4.7, T is almost uniform and by Theorem 4.6, $T^{(2)}, T^{(3)}$ are uniform. □

We remark that any improvement in the size of $T^{(3)}$ would lead to at least a slight improvement in the size of the division circuit.

Acknowledgments. The authors wish to point out that several corrections and substantial clarifications were made by the reviewers. We are especially indebted to one reviewer's very careful reading of this paper.

REFERENCES

[1] P. BEAME, S. COOK, AND H. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.
[2] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.
[3] S. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.
[4] D. KNUTH, *The Art of Computer Programming*, Addison–Wesley, Reading, MA, 1969.
[5] J. REIF, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231–242.
[6] W. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.
[7] J. SERRE, *A Course in Arithmetic*, Springer-Verlag, Berlin, New York, 1973.
[8] R. TANAKA AND N. SZABO, *Residue Arithmetic and Its Application to Computer Technology*, McGraw–Hill, New York, 1968.
[9] I. M. VINAGRADOV, *Elements of Number Theory*, Dover, New York, 1954, Exercise 5b, p. 122
[10] T. V. VU, *Efficient implementation of the Chinese remainder theorem for sign detection and residue decoding*, IEEE Trans. Comput., 34 (1985), pp. 646–651.
[11] I. WEGENER, *The Complexity of Boolean Functions*, Wiley–Teubner, Stuttgart, FRG, 1987.

SIAM J. COMPUT.
Vol. 20, No. 4, pp. 766–805, August 1991
© 1991 Society for Industrial and Applied Mathematics
012

EQUALITY-TEST AND IF-THEN-ELSE ALGEBRAS: AXIOMATIZATION AND SPECIFICATION*

DON PIGOZZI†

Abstract. An *equality-test algebra* has a two-element Boolean sort and an equality-test operation eq_s for each non-Boolean sort s, where $eq_s(x, y)$ equals TRUE if $x = y$ and FALSE otherwise. An *if-then-else algebra* is an equality-test algebra with the if-then-else operations $[_, _, _]_s$ adjoined: $[b, x, y]_s$ equals x if b = TRUE and y if b = FALSE. A finite set of axioms for the conditional-equational (i.e., quasi-equational) theory of equality-test algebras is given. A finite axiomatization of the equational theory of if-then-else algebras is also given, and it is shown that this also serves as a basis for the conditional-equational theory of if-then-else algebras. Finite bases for the equational theories of several classes of algebras closely related to if-then-else algebras were previously known. The power of conditional and equational specifications of equality-test and if-then-else data types are investigated, and the following results, among others, are obtained. (i) Every equality-test data type that can be specified in either the initial or final algebra sense by a finite set of universal first-order sentences can be *completely* specified (i.e., in both the initial and final algebra senses simultaneously) by a finite set of conditional equations. (ii) The same as (i) but with "equality-test" and "conditional equations" replaced, respectively, by "if-then-else" and "equations." (iii) An arbitrary data type that can be specified in the initial algebra sense by a finite set of universal sentences can be specified in the same sense by a finite set of conditional equations with the equality-test operations as hidden operations. (iv) The same as (iii) but with "conditional equations" replaced by "equations," and the if-then-else operations adjoined as additional hidden operations; this holds however only under the additional hypothesis that the original specification is complete.

Key words. equality test, if-then-else, equational logic, conditional equations, quasi equations, variety, quasi variety, specification, initial algebra, final algebra, computable, semicomputable

AMS(MOS) subject classifications. primary 68Q65, 08B05; secondary 68P05, 08A05, 08C15

Introduction. Initial algebra semantics and final algebra semantics have both proved to be important ways of specifying data types.[1] Many of the data types that arise in practice include a two-element Boolean algebra together with equality tests for each sort; we call these *equality-test algebras.* The data type may also include if-then-else operations that select elements of a data domain on the basis of a Boolean test; these are called *if-then-else algebras.* Both kinds of data types turn out to have special properties with regard to both initial and final specification. They allow the data type designer to formulate arbitrary universal first-order axioms as equations in the Boolean sort. This is an important property and is widely used; many of the most natural specification conditions take the form of universal first-order sentences, but only equational and conditional-equational axioms are guaranteed to give an initial specification. The equivalence of a universal first-order formula with its Boolean transform is circumscribed in important ways, however. For example, when specifying stacks of natural numbers we might want to include as an axiom the fact that, if you push the top of a stack on the result of popping the stack, you get the original stack back, provided it is nonempty. Formalizing this in the most natural way we get the universal first-order sentence $\neg(s \approx \mathit{empty}) \rightarrow \mathit{push}(\mathit{pop}(s, x)) \approx s$. If a Boolean sort and equality tests are available the axiom can also be formalized as a Boolean equation: $-eq(s, x) + eq\big(\mathit{push}(\mathit{pop}(s, x))\big) \approx 1$. These two sentences are logically equivalent for equality-test algebras but not for more general algebras. In

* Received by the editors October 26, 1987; accepted for publication May 22, 1989. This research was partially supported by National Science Foundation grant DMS-8805870.
† Iowa State University, Department of Mathematics, Ames, Iowa 50011.
[1] There is a rich literature of initial and final algebra semantics; see § 2 for references.

fact there is no class of algebras that includes the equality-test algebras and is defined exclusively by equations or conditional equations for which the above two sentences are equivalent. However, if a set of universal sentences is an initial specification of an equality-test data type, then the corresponding Boolean transforms also give an initial specification of the data type, provided axioms for the conditional-equational theory of equality-test algebras are adjoined. Moreover, this conditional-equational initial specification is complete in the sense that it is both initial and final. This result reflects an important property of equality-test data types: an initial specification is almost complete in the sense that it can be extended to a complete specification by adding only a single conditional equation; so there is very little difference between initial and final specifications of equality-test data types.

A data type with a recursively enumerable (r.e.) initial, final, or complete specification is, respectively, semicomputable, cosemicomputable, or computable. Thus only computable equality-test data types can have a r.e. initial (or final) specification, and hence every semicomputable or cosemicomputable equality-test data type must in fact be computable.

In the remaining part of the Introduction we summarize the main results of the paper in more detail.

Following Bergstra and Tucker [3] we define a *data structure* to be a heterogeneous algebra in which every element is denoted by a *ground term*, i.e., a term without variables. A data type is an isomorphism class of a data structure. A set Γ of axioms is an *initial* (*final*) *specification* of a data type $\mathbf{A}$ if Γ is a set of sentences of some formal language describing $\mathbf{A}$, and $\mathbf{A}$ is the initial (final or terminal) object in the category of models of Γ. Alternatively, Γ is an initial specification of $\mathbf{A}$ if $\mathbf{A}$ is a model of Γ and every ground identity of $\mathbf{A}$ is a logical consequence of Γ; final specifications can be similarly characterized. Γ is an (initial or final) specification of $\mathbf{A}$ *with hidden sorts and operations* if it is a specification of some enrichment of $\mathbf{A}$ obtained by adjoining new sorts and operations. A specification is called *universal, conditional*, or *equational* if Γ is, respectively, a set of universal first-order sentences, conditional equations (quasi equations), or equations. In this paper we axiomatize the conditional-equational theory of algebras with equality tests and the equational theory of algebras with equality tests and if-then-else operations. We investigate the power of conditional and equational specifications of data types with equality tests and if-then-else operations. We also investigate the power of conditional and equational specifications of arbitrary data types when the Boolean sort and the equality-tests and if-then-else operations are hidden.

Our main results on specification are the following. An *equality-test algebra* is a heterogeneous algebra with a two-element Boolean sort and an *equality-test operation* eq_s for each non-Boolean sort s: $eq_s(x, y)$ takes the value TRUE if $x = y$ and FALSE otherwise. The *if-then-else algebras* are obtained from the equality-test algebras by adjoining the *if-then-else operation* $[_,_,_]_s$ for each non-Boolean sort s: $[\mathrm{TRUE}, x, y]_s = x$ and $[\mathrm{FALSE}, x, y]_s = y$. We give a simple algorithm that converts any universal initial or final specification of an equality-test data type $\mathbf{A}$ into a conditional specification of $\mathbf{A}$; moreover the new specification is *complete* in the sense that it is both initial and final. (See Theorem 3.14.) As a consequence every semicomputable or cosemicomputable equality-test data type is computable (Theorem 3.18). If $\mathbf{A}$ is an arbitrary data type, essentially the same algorithm can be used to convert a universal initial specification of $\mathbf{A}$ into a conditional specification with the equality-tests as hidden operations (Theorem 4.6). In this case the new specification

will be complete only if the original one is complete (Theorem 4.7). Thus an arbitrary data type is computable if and only if its equality-test enrichment is semicomputable or, equivalently, cosemicomputable (Theorem 4.9). Bergstra and Tucker [1]–[3] have systematically studied the relationship between the computability of arbitrary data types and their specification. See § 3 for more details.

Many of the above results can be improved in the presence of if-then-else operations. Every universal initial or final specification of an if-then-else data type can be converted to a complete specification by (unconditional) equations (Theorem 5.9). Also, every universal complete specification gives rise to an equational complete specification when the equality-test and if-then-else operations are both taken to be hidden operations (Theorem 5.13). However, as opposed to the equality-test case, arbitrary incomplete initial specifications cannot be converted in this way to equational specifications by hidden operations.

The main feature of the paper, and the principal tool in obtaining the specification results mentioned above, is a detailed analysis of the structure of the algebras in the quasi variety and variety generated, respectively, by all equality-test and if-then-else algebras of a fixed but arbitrary finite signature. We call the members of this quasi variety and variety *generalized equality-test* (GET) and *generalized if-then-else* (GITE) *algebras*. (In the if-then-else case the generated quasi variety and variety coincide.) In the course of this analysis we obtain natural, finite axiom systems AXGET and AXGITE for the quasi variety of GET algebras and variety of GITE algebras, respectively (Corollaries 3.8 and 5.7). Axiomatizations of various varieties closely related to the variety of GITE algebras have previously been obtained by McKenzie [32], Bloom and Tindell [7], Mekler and Nelson [33], Guessarian and Meseguer [18], and Padawitz [40]. Related axiomatization results in the context of logic programming can also be found in Paul [36], [37]. The characteristic feature of the present paper is the almost exclusive use of model-theoretic techniques. In our view the model-theoretic approach has some distinct advantages in the present context. First of all, it seems better suited to the study of quasi varieties since the proof-theoretic methods of conditional-equational logic are not as efficient as those of ordinary equational logic. Second, it seems better able to exploit the heterogeneous character of the data types that arise most commonly in practice. In these data types the Boolean part is always isolated in a separate sort and is connected with the rest of the structure in a well-defined way via the equality-test and possibly the if-then-else operations. This proves especially useful in trying to understand the structure theory of GET and GITE algebras, which can be viewed as natural extensions of the structure theory of Boolean algebras. In fact our two main structure theorems, Theorems 3.7 and 5.6, can be viewed as analogues of the Stone representation theorem of Boolean algebra. As a consequence the axiom systems AXGET and AXGITE divide naturally into a standard set of axioms for Boolean algebras, and systems of axioms that define the equality-test and if-then-else operations in their role as the connections between the Boolean sort and the non-Boolean sorts of the data type.

The paper is divided into five sections. The first section is preliminary. It reviews the basics of universal algebra, logic, model theory, and Boolean algebra; it also introduces most of the notation and terminology used later. We have made this section more detailed than normally expected since model-theoretic techniques are not so common in the literature of abstract data types, and we wanted to make the paper as self-contained as possible. Section 2 is an outline of the general theory of data specification. Most of the results here are known, but they are presented in

a form most convenient for applications in the later sections. The specification of equality-test algebras is investigated in §3, and in §4 we study the specification of arbitrary data types with equality-tests adjoined as hidden operations. The results of §§3 and 4 are extended to if-then-else algebras in § 5. The last section of the paper contains some final remarks and a brief discussion of possible directions for future research.

1. Preliminaries.

1.1. General algebraic preliminaries. Let S be a set of *sorts*. An *S-sorted signature* Σ is an $S^* \times S$-sorted family $\langle \Sigma_{w,s} : w \in S^*, s \in S \rangle$. Each $\sigma \in \Sigma_{w,s}$ is called an *operation symbol* of *arity* w and *target sort* s; we also write $\sigma : w \to s$ to indicate that $\sigma \in \Sigma_{w,s}$. For simplicity we assume that $\Sigma_{w,s} \cap \Sigma_{w',s'} = \emptyset$ whenever $w \neq w'$ or $s \neq s'$. Operation symbols with arity λ (the empty string) are called *constant symbols.* A *Σ-algebra* $\mathbf{A} = \langle A, \{\sigma^{\mathbf{A}} : \sigma \in \Sigma\} \rangle$ consists of an S-sorted set $A = \langle A_s : s \in S \rangle$, called the *domain set* of $\mathbf{A}$, and a *fundamental operation* $\sigma^{\mathbf{A}} : A^w \to A_s$, for each $\sigma \in \Sigma_{w,s}$, where $A^w = A_{s_0} \times A_{s_1} \times \cdots \times A_{s_{n-1}}$ if $w = s_0 s_1 \cdots s_{n-1}$. If σ is a constant symbol with target s, then $\sigma^{\mathbf{A}} \in A_s$; $\sigma^{\mathbf{A}}$ is called a *constant* of $\mathbf{A}$. It is always assumed that $\bigcup_{s \in S} A_s \neq \emptyset$, i.e., $A_s \neq \emptyset$ for at least one $s \in S$; in this paper we further assume that $A_s \neq \emptyset$ for *all* $s \in S$. While this assumption excludes some data structures of practical interest, it substantially simplifies the metamathematics, in particular the equational metatheory, of heterogeneous algebras in the sense that with very minor exceptions all the results for homogeneous algebras carry over mutatis mutandis to the heterogeneous case.[2] A Σ-algebra is *trivial* if each of its domains contains exactly one element.

In general, algebras will be denoted by boldface letters and their domain sets by the corresponding lightface letters. Subalgebras, homomorphisms, isomorphisms, congruence relations, quotient algebras, and Cartesian products are defined in the usual way. We write $\mathbf{A} \subseteq \mathbf{B}$ to indicate that $\mathbf{A}$ is a subalgebra of $\mathbf{B}$. For any S-sorted subset A' of the domain set A of $\mathbf{A}$ we denote by $\langle A' \rangle_{\mathbf{A}}$ the subalgebra of $\mathbf{A}$ generated by A', i.e., the smallest $\mathbf{B} \subseteq \mathbf{A}$ such that $A' \subseteq B$. By the *minimal subalgebra of* $\mathbf{A}$ we mean the subalgebra generated by the empty S-sorted set, i.e., $\langle \emptyset : s \in S \rangle$; $\mathbf{A}$ itself is called *minimal* if it coincides with its own minimal subalgebra. In view of our assumption that $A_s \neq \emptyset$ for every Σ-algebra $\mathbf{A}$, in order to ensure that minimal subalgebras always exist we assume that Σ has the property that there exist at least one *ground term* of sort s for each $s \in S$ (see below). Let $\mathbf{Min}\,\mathbf{A}$ denote the minimal subalgebra of $\mathbf{A}$, and *Min* $\mathbf{A}$ its domain set. For any class K of Σ-algebras, $\mathsf{Min}\,\mathsf{K} = \{\mathbf{Min}\,\mathbf{A} : \mathbf{A} \in \mathsf{K}\}$.

We write $h : \mathbf{A} \to \mathbf{B}$ to indicate that h is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$; h is an S-sorted family $\langle h_s : s \in S \rangle$ of functions, where $h_s : A_s \to B_s$ for each $s \in S$. If $\mathbf{C}$ is a subalgebra of $\mathbf{A}$, the restriction $h|C$ of h to the domain set of $\mathbf{C}$ is also a homomorphism from $\mathbf{C}$ to $\mathbf{B}$. We normally write h for $h|C$ when $\mathbf{C}$ is clear from context. The *image of C under h*, $h(C) = \langle h(C)_s : s \in S \rangle$ where $h(C)_s = \{h(c) : c \in C_s\}$, is closed under the fundamental operations of $\mathbf{B}$, and thus forms the domain of a subalgebra that we denote by $h(\mathbf{C})$. h is *surjective* if and only if $h(\mathbf{A}) = \mathbf{B}$; it is *injective* if and only if $h : \mathbf{A} \to h(\mathbf{A})$ is an isomorphism. If $h : \mathbf{A} \to \mathbf{B}$ is any homomorphism of Σ-algebras, then $h : \mathbf{Min}\,\mathbf{A} \to \mathbf{Min}\,\mathbf{B}$ is always a surjective

[2] The equational metatheory of the more general class of heterogeneous algebras is developed in Goguen and Meseguer [14]; see also Ehrig and Mahr [11]. All the results of the present paper can be extended in this context with appropriate changes.

homomorphism. The equivalence relation of isomorphism between algebras is denoted by $\cong$.

Congruence relations will be represented by the symbol $\equiv$ and by capital Greek letters Θ, Φ, Θ_0, $\Phi_0, \cdots$. Let $\Theta = \langle \Theta_s : s \in S \rangle$ be a congruence on $\mathbf{A}$. The fact that two elements a and b of A_s are in the relation Θ can be expressed in any one of the following ways: $\langle a, b \rangle \in \Theta_s$, $a \equiv b \pmod{\Theta_s}$, $a \equiv b\ (\Theta_s)$, or $a \equiv_{\Theta_s} b$. We often omit the sort subscript s when no confusion is likely. This applies in all other situations where a specific sort is appropriate.

Every homomorphism $h : \mathbf{A} \to \mathbf{B}$ induces a congruence Θ on $\mathbf{A}$ defined by $\Theta_s = \{\langle a, b \rangle : a, b \in A_s, h_s(a) = h_s(b)\}$ for every $s \in S$; Θ is called the *relation kernel of h*. The *quotient of* $\mathbf{A}$ by the congruence Θ is denoted by $\mathbf{A}/\Theta$. $[a]_\Theta$ is the equivalence class of $a \in A_s$ under Θ, and $A/\Theta = \langle (A/\Theta)_s : s \in S \rangle$, where $(A/\Theta)_s = \{[a]_\Theta : a \in A_s\}$ for each $s \in S$, is the domain set of $\mathbf{A}/\Theta$. The quotient map $h : A \to A/\Theta$ such that $h(a) = [a]_\Theta$ for each element a of $\mathbf{A}$ is called the *natural map*; it is a surjective homomorphism with Θ as its relation kernel. If $f : \mathbf{A} \to \mathbf{B}$ is a surjective homomorphism and Θ is its relation kernel, then $\mathbf{A}/\Theta \cong \mathbf{B}$; in fact $f \circ h^{-1}$ is an isomorphism between $\mathbf{A}/\Theta$ and $\mathbf{B}$, where h is the natural map from $\mathbf{A}$ to $\mathbf{A}/\Theta$.

The set of all congruences on $\mathbf{A}$ is denoted by *Con* $\mathbf{A}$. They form a complete lattice $\mathbf{Con}\,\mathbf{A}$ under the relation of set-theoretical inclusion $\Theta \subseteq \Phi$ between S-sorted sets (i.e., $\Theta_s \subseteq \Phi_s$ for all $s \in S$). The greatest lower bound of any system Θ_i, for $i \in I$, of congruences of $\mathbf{A}$ is the sorted set-theoretical intersection $\bigcap_{i \in I} \Theta_i = \langle \bigcap_{i \in I} \Theta_{i,s} : s \in S \rangle$. The least upper bound $\bigvee_{i \in I} \Theta_i$ does not in general coincide with the sorted set-theoretical union. It does, however, when $\{\Theta_i : i \in I\}$ is directed by inclusion: for all i and j there exists a k such that $\Theta_i \cup \Theta_j \subseteq \Theta_k$ (i.e., $\Theta_{i,s} \cup \Theta_{j,s} \subseteq \Theta_{k,s}$ for all $s \in S$). The smallest congruence, the identity relation on each sort, is denoted by Δ_A; the largest, the universal relation on each sort, is denoted by ∇_A. $\mathbf{A}$ is called *simple* if $\mathbf{A}$ is nontrivial and Δ_A and ∇_A are its only congruences. Thus $\mathbf{A}$ is simple if and only if it is nontrivial and isomorphic to each of its nontrivial homomorphic images.

The following *correspondence theorem* between the congruences on an algebra and its quotient holds for heterogeneous algebras just as it does for homogeneous algebras: for any Σ-algebra $\mathbf{A}$ and congruence Θ on $\mathbf{A}$, the mapping $\Phi \mapsto \{\langle [a]_\Theta, [b]_\Theta \rangle : \langle a, b \rangle \in \Phi\}$ is a lattice isomorphism between the sublattice $\{\Phi : \Theta \subseteq \Phi \in \mathit{Con}\,\mathbf{A}\}$ of $\mathbf{Con}\,\mathbf{A}$ and the lattice $\mathbf{Con}(\mathbf{A}/\Theta)$.

The *Cartesian* or *direct product* of any system $\langle \mathbf{A}_i : i \in I \rangle$ of Σ-algebras is written $\prod_{i \in I} \mathbf{A}_i$; its domain set is $\prod_{i \in I} A_i = \big\langle \prod_{i \in I} A_{i,s} : s \in S \big\rangle$. If I is the empty set, then $\prod_{i \in I} \mathbf{A}_i$ is by definition a trivial algebra. The *projection* $\pi_i : \prod_{i \in I} A_i \to A_i$ is a surjective homomorphism from $\prod_{i \in I} \mathbf{A}_i$ onto $\mathbf{A}_i$. If all the factors $\mathbf{A}_i$ coincide with the same algebra $\mathbf{A}$, then $\prod_{i \in I} \mathbf{A}_i$ is called a *Cartesian* or *direct power* of $\mathbf{A}$, and is written $\mathbf{A}^I$.

$\mathbf{B}$ is a *subdirect product* of a system $\langle \mathbf{A}_i : i \in I \rangle$ of algebras if $\mathbf{B} \subseteq \prod_{i \in I} \mathbf{A}_i$ and $\pi_i(\mathbf{B}) = \mathbf{A}_i$ for each $i \in I$; symbolically, this is expressed by writing $\mathbf{B} \subseteq_{\mathrm{SD}} \prod_{i \in I} \mathbf{A}_i$. It is easy to see that $\mathbf{Min}\big(\prod_{i \in I} \mathbf{A}_i\big) \subseteq_{\mathrm{SD}} \prod_{i \in I} \mathbf{Min}\,\mathbf{A}_i$ for every system $\mathbf{A}_i$, with $i \in I$, of Σ-algebras. An injective homomorphism $h : \mathbf{C} \to \prod_{i \in I} \mathbf{A}_i$ is called a *subdirect representation* of $\mathbf{C}$ in a class K of algebras if $h(\mathbf{C}) \subseteq_{\mathrm{SD}} \prod_{i \in I} \mathbf{A}_i$ and $\mathbf{A}_i \in \mathsf{K}$ for all $i \in I$. In this event $\bigcap_{i \in I} \Theta_i = \Delta_C$ where Θ_i is the relation kernel of $\pi \circ h$. To see this observe that, for any sort s and any $c_0, c_1 \in C_s$, $c_0 \equiv c_1\big(\bigcap_{i \in I} \Theta_{i,s}\big)$ if and only if $c_0 \equiv c_1(\Theta_{i,s})$ for all $i \in I$ if and only if $\pi_{i,s}(h(c_0)) = \pi_{i,s}(h(c_1))$ for all $i \in I$ if and only if $c_0 = c_1$. Conversely, if Θ_i, $i \in I$, is any system of congruences of $\mathbf{C}$ such

that $\bigcap_{i\in I}\Theta_i = \Delta_C$, then it is easy to check that the mapping $h : C \to \prod_{i\in I} C_i/\Theta_i$ defined by $h(c) = \langle [c]_{\Theta_i} : i \in I\rangle$ is a subdirect representation of $\mathbf{C}$ by the system of quotient algebras $\langle \mathbf{C}/\Theta_i : i \in I\rangle$.

An S'-sorted signature Σ' is an *enrichment* of the S-sorted signature Σ if $S \subseteq S'$ and $\Sigma_{w,s} \subseteq \Sigma'_{w,s}$ for all $w \in S^*$ and $s \in S$. If $\mathbf{A}$ is a Σ'-algebra, then $\mathbf{A}|_\Sigma$ is the Σ-algebra defined by $(\mathbf{A}|_\Sigma)_s = \mathbf{A}_s$ for $s \in S$, and $\sigma^{\mathbf{A}|_\Sigma} = \sigma^{\mathbf{A}}$ for $\sigma \in \Sigma_{w,s}$ with $w \in S^*$ and $s \in S$. $\mathbf{A}|_\Sigma$ is called the *Σ-reduct of* $\mathbf{A}$, and $\mathbf{A}$ is called a *Σ'-enrichment of* $\mathbf{A}|_\Sigma$. $\mathbf{B}$ is a *subreduct of* $\mathbf{A}$ if $\mathbf{B} \subseteq \mathbf{A}|_\Sigma$. For example, $\mathbf{Min}\,(\mathbf{A}|_\Sigma)$ is always a subreduct of $\mathbf{Min}\,\mathbf{A}$, but is not in general a reduct.

1.2. Metamathematical preliminaries. The set of all *Σ-terms* is denoted by T_Σ. Thus T_Σ is the S-sorted set $\langle T_{\Sigma,s} : s \in S\rangle$ defined by simultaneous recursion in the following way: $\Sigma_{\lambda,s} \subseteq T_{\Sigma,s}$ for each $s \in S$, and, if $\sigma \in \Sigma_{w,s}$ with $w = s_0 s_1 \cdots s_{n-1}$, and $t_{s_i} \in T_{\Sigma,s_i}$ for each i, then $\sigma(t_{s_0},\cdots,t_{s_{n-1}}) \in T_{\Sigma,s}$. By our basic assumption about Σ, we have $T_{\Sigma,s} \neq \emptyset$ for each $s \in S$. The *Σ-term algebra* $\mathbf{T}_\Sigma$ is the Σ-algebra with domain set T_Σ and fundamental operations $\sigma^{\mathbf{T}_\Sigma}(t_{s_0},\cdots,t_{s_{n-1}}) = \sigma(t_{s_0},\cdots,t_{s_{n-1}})$. Let $X = \langle X_s : s \in s\rangle$ be an S-sorted family of sets of variable symbols such that X and Σ have no symbols in common. We define $T_\Sigma(X) = T_{\Sigma'}$ where Σ' is the S-sorted signature with $\Sigma'_{\lambda,s} = \Sigma_{\lambda,s} \cup X_s$ for all $s \in S$, and $\Sigma'_{w,s} = \Sigma_{w,s}$ for $w \in S^+$ and $s \in S$. Terms in $T_\Sigma(X)$ are called *(Σ, X)-terms* or just *Σ-terms* when X is clear from context. We usually refer to terms in T_Σ as *ground Σ-terms* to emphasize the fact that they contain no variable symbols. The *(Σ, X)-term algebra* $\mathbf{T}_\Sigma(X)$ has domain set $T_\Sigma(X)$ and fundamental operations defined in the usual way. Its minimal subalgebra is $\mathbf{T}_\Sigma$, the Σ-algebra of ground terms.

$\mathbf{T}_\Sigma(X)$ is an absolutely free Σ-algebra in the sense that, for any Σ-algebra $\mathbf{A}$, and any S-sorted function $f : X \to A$ (i.e., $f = \langle f_s : s \in S\rangle$ where $f_s : X_s \to A_s$), there exists a unique homomorphism $h : \mathbf{T}_\Sigma(X) \to \mathbf{A}$ such that $h_s|X_s = f_s$ for each $s \in S$. For each $t(x_0,\cdots,x_{n-1}) \in T_\Sigma(X)$ and each assignment $a_0,\cdots,a_{n-1}$ of elements of $\mathbf{A}$ (of the appropriate sorts) to the variables of t, we define $t^{\mathbf{A}}(a_0,\cdots,a_{n-1}) = h(t)$ where $h : \mathbf{T}_\Sigma(X) \to \mathbf{A}$ is any homomorphism such that $h(x_i) = a_i$ for all i. In particular, we get a unique homomorphism $h_{\mathbf{A}} : \mathbf{T}_\Sigma \to \mathbf{A}$ for each Σ-algebra $\mathbf{A}$, i.e., $\mathbf{T}_\Sigma$ is *initial* in the class of all Σ-algebras. Clearly the image $h_{\mathbf{A}}(\mathbf{T}_\Sigma)$ is the minimal algebra of $\mathbf{A}$. This gives an alternative and useful characterization of the minimal subalgebra of $\mathbf{A}$ as the set of all elements of $\mathbf{A}$ denoted by ground terms, i.e., as $\{t^{\mathbf{A}} : t \in T_\Sigma\}$.[3]

Let Σ be a fixed but arbitrary signature; we assume from now on that Σ is always finite in the sense that S and $\bigcup_{w\in S^*,s\in S}\Sigma_{w,s}$ are finite. X is a fixed S-sorted set of variable symbols with X_s a countably infinite set for each $s \in S$. We will represent the variables by $x, y, z, x_0, y_0, \cdots$; we do not usually specify their sort, leaving it to be made clear from context. For each sort $s \in S$, the *Σ-equations of sort s*, or simply the *s-equations*, are the formulas of the form $t \approx r$, where $t, r \in T_{\Sigma,s}(X)$.

The set of *first-order Σ-formulas* are constructed in the usual way from the Σ-equations by means of the primitive propositional connectives $\vee$, $\wedge$, $\neg$, and $\rightarrow$, and the quantifiers $\exists$ and $\forall$. We use lower case roman letters $t, r, p, q, t_0, r_0, \cdots$ for terms, and lower case Greek letters $\varphi, \psi, \vartheta, \varphi_0, \psi_0, \cdots$ for first-order formulas. When we want to indicate the free variables of φ we write it in the form $\varphi(x_0, x_1,\cdots,x_{n-1})$; similarly, writing a term t in the form $t(x_0,\cdots,x_{n-1})$ means that the variables of t are included in the list $x_0,\cdots,x_{n-1}$. A formula without free variables is called a

[3] This is the basis for several alternate expressions that can be found in the literature for minimal algebras, such as *term-generated* and *reachable*.

sentence. For any formula $\varphi(x_0, \cdots, x_{n-1})$ with free variables $x_0, \cdots, x_{n-1}$ and any algebra $\mathbf{A}$ and elements $a_0, \cdots, a_{n-1}$ of the proper sorts, we write $\mathbf{A} \models \varphi[a_0, \cdots, a_{n-1}]$ to indicate that φ is true in $\mathbf{A}$ when $x_0, \cdots, x_{n-1}$ are interpreted, respectively, as $a_0, \cdots, a_{n-1}$. In particular, if φ is the equation $t(x_0, \cdots, x_{n-1}) \approx r(x_0, \cdots, x_{n-1})$, then $\mathbf{A} \models \varphi[a_0, \cdots, a_{n-1}]$ if and only if $t^{\mathbf{A}}(a_0, \cdots, a_{n-1}) = r^{\mathbf{A}}(a_0, \cdots, a_{n-1})$, where $t^{\mathbf{A}}(a_0, \cdots, a_{n-1})$ and $r^{\mathbf{A}}(a_0, \cdots, a_{n-1})$ are the elements of $\mathbf{A}$ obtained when the terms t and r are evaluated in $\mathbf{A}$ with $x_0, \cdots, x_{n-1}$ interpreted, respectively, as $a_0, \cdots, a_{n-1}$. More precisely, $t^{\mathbf{A}}(a_0, \cdots, a_{n-1}) = h(t)$ where h is a homomorphism from $\mathbf{T}_\Sigma(X)$ into $\mathbf{A}$ such that $h(x_i) = a_i$ for $i = 0, \cdots, n-1$. If t is a ground term, i.e., $t \in T_\Sigma$, then t has a unique evaluation $t^{\mathbf{A}}$ by means of the unique homomorphism $h : \mathbf{T}_\Sigma \to \mathbf{A}$. Similarly, if φ is the conditional equation

$$\bigwedge_{i<m} t_i(x_0, \cdots, x_{n-1}) \approx r_i(x_0, \cdots, x_{n-1}) \ \rightarrow\ p(x_0, \cdots, x_{n-1}) \approx q(x_0, \cdots, x_{n-1}),$$

then $\mathbf{A} \models \varphi[a_0, \cdots, a_{n-1}]$ holds if and only if $t_i^{\mathbf{A}}(a_0, \cdots, a_{n-1}) \neq r_i^{\mathbf{A}}(a_0, \cdots, a_{n-1})$ for at least one $i < m$, or $p^{\mathbf{A}}(a_0, \cdots, a_{n-1}) = q^{\mathbf{A}}(a_0, \cdots, a_{n-1})$.

For any sentence φ, $\mathbf{A} \models \varphi$ means φ is true in $\mathbf{A}$. In this situation we say that $\mathbf{A}$ is a *model of* or *satisfies* φ, or that φ is *valid in* $\mathbf{A}$. $\mathbf{A} \models \Gamma$ for a set of sentences Γ means that $\mathbf{A}$ is a model of each sentence in Γ. We define $\mathsf{Mod}_\Sigma\Gamma = \{\, \mathbf{A} : \mathbf{A}$ a Σ-algebra such that $\mathbf{A} \models \Gamma \,\}$. $\mathsf{Mod}_\Sigma\Gamma$ is called the *model class of* Γ. For any set Γ of sentences and any sentence φ, we write $\Gamma \models \varphi$ when φ is a *logical consequence of* Γ, i.e., φ is valid in every model of Γ.

We write $\Gamma \models_{\mathrm{Min}} \varphi$ to mean that φ is true in every minimal algebra that is a model of Γ. The special consequence relation $\models_{\mathrm{Min}}$ can be obtained from the ordinary consequence relation $\models$ by adding the structural induction axiom of Burstall [10] as a new implicit premise; more precisely, if φ is the structural induction axiom, then for any first-order sentence ψ we have $\Gamma \models_{\mathrm{Min}} \psi$ if and only if $\Gamma, \varphi \models \psi$. The structural induction axiom cannot be formulated in first-order logic, and there is a great difference between $\models_{\mathrm{Min}}$ and $\models$. For example, if $\Sigma = \{\, 0, succ, +, \cdot \,\}$, the signature of the natural numbers with successor, addition, and multiplication, then the structural induction axiom is equivalent to the usual second-order Peano induction axiom.

We write $\Gamma \vdash \varphi$ to mean that φ is formally deducible from Γ by any one of the standard set of axioms and rules of inference of first-order predicate logic. By the Gödel completeness theorem, $\models$ and $\vdash$ define the same relation between sets of sentences and single sentences. Thus $\models$ is a recursively enumerable (r.e.) relation (under an appropriate Gödel numbering), and for any r.e. set Γ, $\{\, \varphi : \Gamma \models \varphi \,\}$ is r.e. However, $\models_{\mathrm{Min}}$ is not r.e. If Γ is the set of first-order Peano axioms, then $\{\, \varphi : \Gamma \models_{\mathrm{Min}} \varphi \,\}$ is the set of all true sentences of arithmetic (i.e., true of the natural numbers), and is not r.e. (in fact it is hyperarithmetical).

Every formula $\varphi(x_0, \cdots, x_{n-1})$ with free variables $x_0, \cdots, x_{n-1}$ is logically equivalent to one of the form $Q_0 y_0 Q_1 y_1 \cdots Q_{m-1} y_{m-1} \psi(x_0, \cdots, x_{n-1}, y_0, \cdots, y_{m-1})$, where ψ contains no quantifiers; this is called the *prenex normal form* of φ, and there is a simple algorithm for constructing it. A sentence is said to be *universal* if all quantifiers in its prenex form are universal. Two special kinds of universal sentences are the *universally closed equations*

$$\forall x_0 \cdots \forall x_{n-1} \big(t(x_0, \cdots, x_{n-1}) \approx r(x_0, \cdots, x_{n-1}) \big),$$

and the *universally closed conditional equations*, or *quasi equations*,

$$\forall x_0 \cdots \forall x_{n-1} \Big(\bigwedge_{i<m} t_i(x_0, \cdots, x_{n-1}) \approx r_i(x_0, \cdots, x_{n-1}) \\ \rightarrow p(x_0, \cdots, x_{n-1}) \approx q(x_0, \cdots, x_{n-1}) \Big).$$

When used as axioms, in particular in the specification of data structures, the quantifiers are normally omitted to simplify notation, and the sentences referred to simply as *equations* and *conditional equations*, respectively.

Of special interest are equations of the form $t \approx r$, where t and r are ground terms (i.e., terms without variables). They are called *ground* Σ*-equations*. Any ground equation such that $\mathbf{A} \models t \approx r$, i.e., $t^{\mathbf{A}} = r^{\mathbf{A}}$, is called a *ground identity of* $\mathbf{A}$. If $\mathbf{A}$ is a minimal algebra and $\mathbf{B}$ is any Σ-algebra, then there exists a (necessarily unique) homomorphism from $\mathbf{A}$ into $\mathbf{B}$ if and only if $\mathbf{A} \models t \approx r$ implies $\mathbf{B} \models t \approx r$ for each ground equation $t \approx r$, i.e., every ground identity of $\mathbf{A}$ is a ground identity of $\mathbf{B}$. In this case the unique homomorphism is given by $h(t^{\mathbf{A}}) = t^{\mathbf{B}}$ for every $t \in T_\Sigma$.

Let K be any class of Σ-algebras. K is an *elementary class* if $\mathsf{K} = \mathsf{Mod}\,\Gamma$ for some set Γ of first-order sentences. K is *universal* if it is the model class of a set of universal sentences. K is called a *variety*, respectively, a *quasi variety* or *conditional class*, if it is the model class of a set of equations, respectively, conditional equations or quasi equations. These three classes of algebras can also be specified by algebraic means. Let I be any nonempty set. By a *filter* on I we mean any family F of subsets of I with the following properties: (1) $I \in F$; (2) if $J \in F$ and $J \subseteq K \subseteq I$, then $K \in F$; (3) if $J, K \in F$, then $J \cap K \in F$. The following three conditions are equivalent for any filter F: (4) $\emptyset \notin F$; (5) $F \neq 2^I$; (6) not both $X \in F$ and $\overline{X} \in F$ for every $X \subseteq I$ ($\overline{X}$ is the complement of X in I). Any filter satisfying these conditions is called *proper*. The following two conditions are also equivalent for every proper filter F: (7) there is no proper filter on I strictly including F; (8) either $X \in F$ or $\overline{X} \in F$ for every $X \subseteq I$. Any proper filter with these properties is called an *ultrafilter*. It can be shown with the aid of Zorn's lemma that every proper filter can be extended to an ultrafilter; in fact, given any $J \subseteq I$ such that $J \notin F$, there exists an ultrafilter U such that $F \subseteq U$ and $J \notin U$. Thus any proper filter is the intersection of all ultrafilters that include it.

Let $\langle \mathbf{A}_i : i \in I \rangle$ be a system of Σ-algebras and F a filter on I. The S-sorted binary relation Θ_F on $\prod_{i \in I} A_i$ is defined by the condition that, for all $s \in S$ and all $\langle a_i : i \in I \rangle, \langle b_i : i \in I \rangle \in \prod_{i \in I} A_{i,s}$,

$$\big\langle \langle a_i : i \in I \rangle, \langle b_i : i \in I \rangle \big\rangle \in \Theta_{F,s} \quad \text{iff} \quad \{\, i \in I : a_i = b_i \,\} \in F.$$

It is easy to check that Θ_F is a congruence. The quotient algebra $\big(\prod_{i \in i} \mathbf{A}_i\big)/\Theta$, where F is any proper or improper filter on I, is called a *reduced product of* $\langle \mathbf{A}_i : i \in I \rangle$, and it is called an *ultraproduct* if F is an ultrafilter.

THEOREM 1.1. *Let φ be any first-order sentence.*

(i) (Łoś [26]) *If $\mathbf{A}_i \models \varphi$ for each $i \in I$, and U is an ultrafilter on I, then $\big(\prod_{i \in I} \mathbf{A}_i\big)/\Theta_F \models \varphi$.*

(ii) *If φ is a universal sentence, $\mathbf{A} \models \varphi$, and $\mathbf{B} \subseteq \mathbf{A}$, then $\mathbf{B} \models \varphi$.*

(iii) *If φ is a conditional equation (in particular, an equation), $\mathbf{A}_i \models \varphi$ for each $i \in I$, and F is any filter on I, then $\big(\prod_{i \in I} \mathbf{A}_i\big)/\Theta_F \models \varphi$.*

(iv) *If ϑ is an equation, $\mathbf{A} \models \vartheta$, and $\mathbf{B}$ is a homomorphic image of $\mathbf{A}$, then $\mathbf{B} \models \vartheta$.*

Part (i) is commonly referred to as Łoś's theorem.

These properties essentially serve to characterize the notions of universal class, quasi variety, and variety. For any class K of algebras let $\mathbf{I}\,\mathsf{K}$, $\mathbf{H}\,\mathsf{K}$, $\mathbf{S}\,\mathsf{K}$, $\mathbf{P}\,\mathsf{K}$, $\mathbf{P}_{\mathrm{SD}}\mathsf{K}$, $\mathbf{P}_{\mathrm{R}}\mathsf{K}$, and $\mathbf{P}_{\mathrm{U}}\mathsf{K}$, respectively, be the class of all isomorphic images, subalgebras, homomorphic images, products, subdirect products, reduced products, and ultraproducts of members of K. For a single algebra $\mathbf{A}$ we write $\mathbf{IA}, \mathbf{HA}, \cdots$ instead of $\mathbf{I}\{\mathbf{A}\}, \mathbf{H}\{\mathbf{A}\}, \cdots$. Since any product reduced by an improper filter gives a trivial algebra, $\mathbf{P}_{\mathrm{R}}\,\mathsf{K}$ always contains the trivial algebras.

The three parts of the following theorem are due, respectively, to Łoś [26], Mal'cev [29], and Birkhoff [6]; for the proof, see Grätzer [17] and Mal'cev [30].

THEOREM 1.2. *Let* K *be any class of* Σ*-algebras closed under isomorphism, i.e.,* $\mathbf{I}\,\mathsf{K} = \mathsf{K}$.

(i) K *is a universal class if and only if* $\mathbf{S}\,\mathsf{K} = \mathsf{K}$ *and* $\mathbf{P}_{\mathrm{U}}\mathsf{K} = \mathsf{K}$.

(ii) K *is a quasi variety if and only if* $\mathbf{S}\,\mathsf{K} = \mathsf{K}$ *and* $\mathbf{P}_{\mathrm{R}}\mathsf{K} = \mathsf{K}$.

(iii) K *is a variety if and only if* $\mathbf{H}\,\mathsf{K} = \mathsf{K}$, $\mathbf{S}\,\mathsf{K} = \mathsf{K}$, *and* $\mathbf{P}\,\mathsf{K} = \mathsf{K}$.

Using Łoś's theorem we get as a corollary of part (i) that, if K is an elementary class, then K is universal if and only if it is closed under the formation of subalgebras.

By the *universal theory* of K we mean the set of all universal sentences true in K; similarly, the *conditional theory* and the *equational theory* are, respectively, the sets of all conditional equations and equations that hold identically in K. By the *quasi variety* and *variety generated by* K, in symbols $\mathsf{Qv}\,\mathsf{K}$ and $\mathsf{Va}\,\mathsf{K}$, we mean the intersection of all quasi varieties and varieties, respectively, including K; $\mathsf{Qv}\,\mathsf{K}$ can also be characterized as $\mathsf{Mod}\,\Gamma$ where Γ is the conditional theory of K. The *variety generated by* K is characterized analogously.

COROLLARY 1.3. *Let* K *be any class of algebras.*

(i) $\mathsf{Qv}\,\mathsf{K} = \mathbf{ISP}_{\mathrm{R}}\mathsf{K}$.

(ii) $\mathsf{Va}\,\mathsf{K} = \mathbf{HSP}\,\mathsf{K}$.

1.3. Boolean algebra. The class BA of *Boolean algebras* is the variety of homogeneous algebras defined by the following equational axioms:

$$
\begin{aligned}
(x+y)+z \approx x+(y+z), &\qquad (x\cdot y)\cdot z \approx x\cdot(y\cdot z),\\
x+y \approx y+x, &\qquad x\cdot y \approx y\cdot x,\\
x\cdot(y+z) \approx x\cdot y + x\cdot z, &\qquad x + y\cdot z \approx (x+y)\cdot(x+z),\\
x + -x \approx, &\qquad x\cdot -x \approx 0,\\
x+0 \approx x, &\qquad x\cdot 1 \approx x,\\
x+1 \approx 1, &\qquad x\cdot 0 \approx 0.
\end{aligned}
$$

We denote the *two-element* Boolean algebra by $\mathbf{B}_2$. Let $\mathbf{A} = \langle A, +, \cdot, -, 0, 1\rangle$ be a Boolean algebra. $a \leq b$ if and only if $a \cdot b = a$ (equivalently, $a + b = b$) defines a lattice ordering on A, where one is the largest element, zero is the smallest, $a + b$ is the least upper bound of a and b, and $a \cdot b$ is the greatest lower bound.

By the well-known theorem of Stone [41] every Boolean algebra is isomorphic to a field of subsets of some set I under set-theoretical union, intersection, and complementation relative to I. An essentially equivalent and more algebraic version of this result is that every Boolean algebra is isomorphic to a subdirect product of two-element Boolean algebras, or, in symbolic form, $\mathsf{BA} = \mathbf{IP}_{\mathrm{SD}}\mathbf{B}_2$.

Let $\mathbf{B}$ be an arbitrary Boolean algebra. By a *filter* on $\mathbf{B}$ we mean a subset F of B satisfying the following conditions: (1) $1 \in F$; (2) if $a \in F$ and $a \leq b$, then

$b \in F$; (3) if $a, b \in B$, then $a \cdot b \in B$. F is *proper* if $F \neq B$ (equivalently, $0 \notin F$, or F does not contain both a and $-a$ for any $a \in B$). F is *maximal*, or *prime*, if F is proper and is included in no strictly larger proper filter (equivalently, F contains either a or $-a$ for every $a \in B$). By Zorn's lemma every proper filter is included in a prime filter, and, in fact, is the intersection of all prime filters that include it. The set of all filters of $\mathbf{B}$ forms a complete lattice $\mathbf{Fil\,B}$ under set-theoretical inclusion. The smallest element is the one-element filter $\{1\}$, and the largest is the improper filter B.

There is a bijection between filters and congruences of $\mathbf{B}$ that can be described as follows. Let $Fi\,\Phi = [1]_\Phi$ for each $\Phi \in Con\,\mathbf{B}$, and $\Theta_F = \{\, \langle a, b\rangle : (-a+b)\cdot(a+-b) \in F\,\}$ for each $F \in Fil\,\mathbf{B}$. It is easy to check that $Fi\,\Phi \in Fil\,\mathbf{B}$ and $\Theta_F \in Con\,\mathbf{B}$. Fi is monotone (preserves inclusion) and $\Theta_{Fi\,\Phi} = \Phi$ for each $\Phi \in Con\,\mathbf{B}$, and $Fi\,\Theta_F = F$ for each $F \in Fil\,\mathbf{B}$. Thus Fi is a lattice isomorphism between $\mathbf{Con\,B}$ and $\mathbf{Fil\,B}$ with inverse $F \mapsto \Theta_F$.

Let P be the set of all prime filters of $\mathbf{B}$. Then $\bigcap_{F \in P} F = \{1\}$. By the lattice isomorphism $\bigcap_{F \in P} \Theta_F = \Delta_B$, and hence $\mathbf{B}$ is subdirectly representable in $\{\, \mathbf{B}/\Theta_F : F \in P\,\}$. Since each $F \in P$ is prime, $\mathbf{B}/\Theta_F \cong \mathbf{B}_2$, the two-element Boolean algebra. Consequently, $\mathbf{B}$ is a subdirect power of $\mathbf{B}_2$; this gives the algebraic version of the Stone representation theorem.

2. Data types and their specification. Following Bergstra and Tucker [3] we call a minimal Σ-algebra, where Σ is a finite signature, a Σ-*data structure*. An *abstract* Σ-*data type* (ADT), or simply a Σ-*data type*, is the isomorphism class of a Σ-data structure, i.e., a class of algebras of the form $\mathbf{I}\mathbf{A}$ for some data structure $\mathbf{A}$. We omit the subscript $_\Sigma$ when the signature is clear from context. As is customary we will not be very careful about distinguishing between a data structure and its associated data type.

DEFINITION 2.1. Let Σ be any (finite) signature. A Σ-data type $\mathbf{A}$ is **computable** if the set of all ground identities of $\mathbf{A}$ is recursive (under some standard Gödel numbering). $\mathbf{A}$ is **semi-computable** if its ground identities are recursively enumerable and **cosemicomputable** if its ground identities are co-r.e.

A data type whose set of ground identities is recursive is often said to have a *decidable word problem.* There are several alternative notions of a computable data type, but they turn out to all be equivalent. See Bergstra and Tucker [3] for a detailed discussion of the topic. See also Meseguer and Goguen [34, Thm. 37].

Data types are usually specified by some set Γ of first-order sentences. The notions of initial and final algebra, when they exist, provide a natural and convenient way of associating a unique data type with Γ.

Let K be any class of Σ-algebras. A Σ-algebra $\mathbf{A}$ is called an *initial algebra of* K if $\mathbf{A} \in \mathsf{K}$ and there exists a unique homomorphism $h : \mathbf{A} \to \mathbf{B}$ for every $\mathbf{B} \in \mathsf{K}$. $\mathbf{A}$ is a *final algebra of* K if $\mathbf{A} \in \mathsf{K}$ and there exists a unique homomorphism $h : \mathbf{B} \to \mathbf{A}$ for every $\mathbf{B} \in \mathsf{K}$. Initial and final algebras may not exist, but if they do they are unique up to isomorphism and thus form a unique data type if minimal. For this reason we usually speak of *the* initial and final algebra of K, and denote them by $\mathbf{In}\,\mathsf{K}$ and $\mathbf{Fn}\,\mathsf{K}$, respectively. The initial algebra $\mathbf{A}$ of K is called *isoinitial* (Bertoni, Mauri, and Mignoli [5]) or *prime* (Makowsky [28]) if the unique homomorphism $h : \mathbf{A} \to \mathbf{B}$ for each nontrivial $\mathbf{B} \in \mathsf{K}$ is injective.

Suppose K and L are classes of Σ-algebras such that $\mathsf{K} \subseteq \mathsf{L}$. If the initial algebra of L exists and is contained in K, then it must also be initial in K; this is a trivial consequence of the definition of initial algebra. The case is similar for final algebras.

These two facts will be used repeatedly in the sequel without further comment.

A set Γ of first-order Σ-sentences can be viewed as specifying two usually distinct (i.e., nonisomorphic) algebras, the initial and final algebras, when they exist, of its model class $\mathsf{K} = \mathsf{Mod}\,\Gamma$. The problem is that, for arbitrary Γ, $\mathbf{In}\,\mathsf{K}$ and $\mathbf{Fn}\,\mathsf{K}$, even when they exist, may not be minimal and hence not data types. For this reason we restrict our attention to $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$, the class of minimal subalgebras of models of Γ. With this restriction both the initial and final algebras have a very simple characterization in terms of ground equations.

LEMMA 2.2. *Let* K *be any class of* Σ*-algebras, and* $\mathbf{A}$ *a* Σ*-algebra. The following conditions are equivalent:*

(i) $\mathbf{A}$ *is initial in* $\mathsf{Min}\,\mathsf{K}$.

(ii) $\mathbf{A} \in \mathsf{Min}\,\mathsf{K}$ *and there is a homomorphism* $h : \mathbf{A} \to \mathbf{B}$ *for each* $\mathbf{B} \in \mathsf{K}$.

(iii) $\mathbf{A} \in \mathsf{Min}\,\mathsf{K}$ *and, for every ground* Σ*-equation* $t \approx r$, $\mathbf{A} \models t \approx r$ *implies* $\mathsf{K} \models t \approx r$.

If $\mathbf{S}\,\mathsf{K} = \mathsf{K}$, *then each of the above conditions is equivalent to*

(iv) $\mathbf{A}$ *is initial in* K.

If $\mathbf{I}\,\mathsf{K} = \mathbf{S}\,\mathsf{K} = \mathbf{P}\,\mathsf{K} = \mathsf{K}$, *then each of the above conditions is equivalent to*

(v) $\mathbf{A}$ *is minimal and, for each ground* Σ*-equation* $t \approx r$, $\mathbf{A} \models t \approx r$ *if and only if* $\mathsf{K} \models t \approx r$.

Proof. Assume $\mathbf{A}$ is initial in $\mathsf{Min}\,\mathsf{K}$. Then $\mathbf{A}$ is minimal, and there exists a homomorphism $h : \mathbf{A} \to \mathbf{Min}\,\mathbf{B}$ for each $\mathbf{B} \in \mathsf{K}$. h is also a homomorphism into $\mathbf{B}$. Thus (i) implies (ii). To see that the implication in the opposite direction holds, consider any $\mathbf{B} \in \mathsf{Min}\,\mathsf{K}$ and let $\mathbf{B}' \in \mathsf{K}$ such that $\mathbf{B} = \mathbf{Min}\,\mathbf{B}'$. By (ii) there is a homomorphism from $\mathbf{A}$ into $\mathbf{B}$. It is unique since $\mathbf{A}$ is minimal, and clearly $h(\mathbf{A}) \subseteq \mathbf{Min}\,\mathbf{B}' = \mathbf{B}$.

To show that (ii) is equivalent to (iii), assume that (ii) holds. Consider any $\mathbf{B} \in \mathsf{K}$, and let $h : \mathbf{A} \to \mathbf{B}$. Then for each ground equation $t \approx r$, $\mathbf{A} \models t \approx r$ implies $t^{\mathbf{A}} = r^{\mathbf{A}}$, which in turn implies $t^{\mathbf{B}} = h(t^{\mathbf{A}}) = h(r^{\mathbf{A}}) = r^{\mathbf{B}}$, and hence $\mathbf{B} \models t \approx r$. So $\mathbf{A} \models t \approx r$ implies $\mathbf{B} \models t \approx r$ for all $\mathbf{B} \in \mathsf{K}$, i.e., $\mathsf{K} \models t \approx r$. Thus (iii) holds. Conversely, if (iii) holds, then, for each $\mathbf{B} \in \mathsf{K}$, $h(t^{\mathbf{A}}) = t^{\mathbf{B}}$ defines a unique homomorphism from $\mathbf{A}$ into $\mathbf{B}$. So (ii) holds.

If $\mathbf{S}\,\mathsf{K} = \mathsf{K}$, then $\mathsf{Min}\,\mathsf{K} \subseteq \mathsf{K}$, and under the hypothesis, it is clear that (i) is equivalent to (iv) and that (iii) implies (v). To prove that (v) implies (iii) when $\mathbf{I}\,\mathsf{K} = \mathbf{S}\,\mathsf{K} = \mathbf{P}\,\mathsf{K}$, we need only show that (v) implies $\mathbf{A} \in \mathsf{Min}\,\mathsf{K}$. Choose for each ground $t \approx r$ such that $\mathsf{K} \not\models t \approx r$, a $\mathbf{C}_i \in \mathsf{K}$ such that $t^{\mathbf{C}_i} \neq r^{\mathbf{C}_i}$. Let $\mathbf{B} = \prod_{i \in I} \mathbf{C}_i$. Then by (v) we have $t^{\mathbf{A}} = r^{\mathbf{A}}$ if and only if $t^{\mathbf{C}_i} = r^{\mathbf{C}_i}$ for all $i \in I$ if and only if $t^{\mathbf{B}} = r^{\mathbf{B}}$. Since $\mathbf{A}$ is minimal, the map $h(t^{\mathbf{A}}) = t^{\mathbf{B}}$ for each $t \in T_\Sigma$ defines an injective homomorphism from $\mathbf{A}$ into $\mathbf{B}$. Thus $\mathbf{A} \cong h(\mathbf{A}) \in \mathbf{ISP}\,\mathsf{K} = \mathsf{K}$. □

An algebra $\mathbf{A}$ satisfying condition (iii) of Lemma 2.2 is *generic in* K *for* the set of ground equations according to Makowsky [28, Def. 2.4].

LEMMA 2.3. *Let* K *be a class of* Σ*-algebras and* $\mathbf{A}$ *any* Σ*-algebra. The following are equivalent:*

(i) $\mathbf{A}$ *is final in* $\mathsf{Min}\,\mathsf{K}$.

(ii) $\mathbf{A} \in \mathsf{Min}\,\mathsf{K}$, *and, for every ground* Σ*-equation* $t \approx r$, $\mathbf{A} \not\models t \approx r$ *implies* $\mathsf{K} \models \neg(t \approx r)$.

Proof. Let $\mathbf{B}$ be a minimal Σ-algebra. As in the proof of the equivalence of Lemma 2.2(ii) and (iii), there exists a (unique) homomorphism $h : \mathbf{B} \to \mathbf{A}$ if and only if $\mathbf{B} \models t \approx r$ implies $\mathbf{A} \models t \approx r$ for every ground equation $t \approx r$. The equivalence of (i) and (ii) follows immediately from the definitions of the various notions

involved. □

If K contains a trivial algebra, then clearly this is the final algebra of K. Since the trivial algebra is normally of little interest, we restrict our attention to the class K_0 of all nontrivial algebras of K when considering final algebras. In particular, when considering the final algebra specified by a set Γ of first-order sentences, we restrict our attention to the class $(\mathsf{Min}(\mathsf{Mod}\,\mathsf{K}))_0$ of nontrivial minimal subalgebras of models of Γ. The class of models of Γ that have a nontrivial minimal subalgebra can be axiomatized relative to Γ by a finite set of universal sentences, in fact by the negation of a finite conjunction of ground equations.

Choose for each sort $s \in S$ a fixed but arbitrary ground s-term g_s; by our basic assumptions about the signature Σ, such a term always exists. Let Ω_Σ be the finite set of ground Σ-equations

$$\Omega_\Sigma \;=\; \{\,\sigma \approx g_s : s \in S, \sigma \in \Sigma_{\lambda,s}\,\} \,\cup\, \{\,\sigma g_{s_0} \cdots g_{s_{n-1}} \approx g_s \,:\, \sigma \in \Sigma_{s_0 \cdots s_{n-1},s}\,\}.$$

Let ω_Σ be the conjunction of all equations in Ω_Σ.

LEMMA 2.4. *For any Σ-algebra* $\mathbf{A}$ *we have* $\mathbf{A} \models \omega_\Sigma$ *if and only if* $\mathbf{Min}\,\mathbf{A}$ *is trivial. Hence, for any set* Γ *of* Σ*-sentences,*

$$(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0 \;=\; \mathsf{Min}(\mathsf{Mod}(\Gamma \cup \{\neg\omega_\Sigma\})).$$

Proof. $\mathbf{Min}\,\mathbf{A}$ is trivial if and only if $t^{\mathbf{A}} = g_s^{\mathbf{A}}$ for every ground term t of sort s. Thus $\mathbf{Min}\,\mathbf{A}$ trivial implies $\mathbf{A} \models \omega_\Sigma$. Conversely, if $\mathbf{A} \models \omega_\Sigma$, it is easy to prove by induction on the length of t that $t^{\mathbf{A}} = g_s^{\mathbf{A}}$ for every ground term t. □

If Γ is a set of conditional equations, i.e., $\mathsf{Mod}\,\Gamma$ is a quasi variety, then the initial algebra of $\mathsf{Mod}\,\Gamma$ always exists, and it must be minimal and hence a data type (Thatcher, Wagner, and Wright [43]). This result has a partial converse. A set Γ of first-order sentences is said to *admit initial algebras* if $\mathsf{Mod}(\Gamma \cup E)$ has an initial algebra for every set of ground equations. If Γ admits initial, minimal models, then Γ is equivalent to a set of conditional equations (Mahr and Makowsky [27], [28]). The final algebra of $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$ may not exist even when Γ is a set of equations.

DEFINITION 2.5. Let Γ be any set of first-order Σ-sentences, and let $\mathbf{A}$ be a Σ-data type.

(i) Γ is a **weak initial specification** of $\mathbf{A}$ if $\mathbf{A}$ is the initial algebra of $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$.
(ii) Γ is an **initial specification** of $\mathbf{A}$ if $\mathbf{A}$ is the initial algebra of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$.
(iii) Γ is a **final specification** of $\mathbf{A}$ if $\mathbf{A}$ is the final algebra of $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$.
(iv) Γ is a **complete specification** of $\mathbf{A}$ if it is both an initial and final specification of $\mathbf{A}$.

Let Γ be a set of first-order Σ'-sentences where Σ' is any enrichment of Σ. Then Γ is a **weak initial**, an **initial**, a **final**, or a **complete specification** of $\mathbf{A}$ **with hidden sorts and operations** if there exists a Σ'-data type $\mathbf{B}$ such that Γ is, respectively, a weak initial, an initial, a final, or a complete specification of $\mathbf{B}$, and $\mathbf{A} \cong \mathbf{B}\,|_\Sigma$.

If Γ specifies $\mathbf{A}$ in any one of the above senses, we say in addition that Γ is, respectively, a *universal*, a *conditional*, or an *equational* specification of $\mathbf{A}$ if Γ is a set of universal sentences, conditional equations, or equations.

We occasionally speak of a specification (in any one of the above senses) of a data structure $\mathbf{A}$, meaning of course its associated data type $\mathsf{I}\mathbf{A}$.

It is well known that $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ and $\mathsf{Va}(\mathsf{Mod}\,\Gamma)$ always have the same initial algebra (see Theorem 2.8 below). Since this initial algebra always exists, every set Γ of first-order sentences is a weak initial specification of a unique data type. Weak

initial specification has a natural characterization in terms of logical consequence (see Theorem 2.11(i) below).

Note that we define Γ to be an initial specification of $\mathbf{A}$ if $\mathbf{A}$ is the initial algebra of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ rather than $\mathsf{Mod}\,\Gamma$. If Γ is not universal, then Γ may be an initial specification of a data type $\mathbf{A}$ without $\mathsf{Mod}\,\Gamma$ having an initial algebra. But if $\mathsf{Mod}\,\Gamma$ does have an initial algebra, then Γ is an initial specification of its minimal subalgebra in the sense of Definition 2.5 (ii). Makowsky [28] has obtained several different characterizations of those sets of first-order sentences that admit initial (but not necessarily minimal initial) algebras.

Complete specifications have the following useful model-theoretic characterization.

THEOREM 2.6. *Let Γ be any set of first-order Σ-sentences, and let $\mathbf{A}$ be a nontrivial Σ-data structure. The following are equivalent:*

(i) *Γ is a complete specification of $\mathbf{A}$;*

(ii) *$(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0 = \mathsf{I}\mathbf{A}$, i.e., $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$ is a data type and contains $\mathbf{A}$;*

(iii) *$\mathbf{A}$ is isoinitial in $\mathsf{Mod}\,\Gamma$.*

Proof. Let $\mathsf{K} = (\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$. Suppose Γ completely specifies $\mathbf{A}$, i.e., $\mathbf{A}$ is both initial and final in K. Then, for each $\mathbf{B} \in \mathsf{K}$, there are unique homomorphisms between $\mathbf{A}$ and $\mathbf{B}$ in both directions. Since the algebras are minimal, the compositions of the two homomorphisms must coincide with the identity functions on A and B. Thus $\mathbf{A} \cong \mathbf{B}$. This shows that (i) implies (ii), and the implication in the opposite direction is obvious. It is also obvious that (ii) implies (iii); the reverse implication follows easily from the minimality of the algebras of K. □

It follows that, under the assumption that Γ is a set of universal sentences, Γ completely specifies $\mathbf{A}$ if and only if $\mathbf{A}$ is an isoinitial algebra of $\mathsf{Mod}\,\mathsf{K}$.

Recall that an algebra is simple if it is nontrivial and isomorphic to every nontrivial homomorphic image.

COROLLARY 2.7. *Any initial or weak initial specification of a simple data type is complete.*

Proof. Any simple algebra that is initial in a class of algebras is clearly isoinitial. □

The notion of initial algebra specification originated with the ADJ group [15], [16] and Zilles and Liskov [25], [49], [50]. Final algebra specification can be traced back to Guttag's thesis [19] (see also [20]) and was formalized in Giarrantana, Gimona, and Montanari [12] and Wand [45] (see also [8], [22]–[24], [47]). The notion of final specification defined in Definition 2.5(iii) is more restrictive and is due to Bergstra and Tucker [2] (see also [34], [35]). The more general notion of final specification corresponds roughly to final specification in the sense of Definition 2.5(iii) restricted to certain specified sorts. For a general discussion of the specification of data types, see Ehrig and Mahr [11] and Meseguer and Goguen [34].

THEOREM 2.8. *Let K be any elementary class of Σ-algebras. If $\mathsf{Min}\,\mathsf{K}$ has an initial algebra, then $\mathsf{Min}\,\mathsf{K}$, $\mathsf{Qv}\,\mathsf{K}$, and $\mathsf{Va}\,\mathsf{K}$ all have the same initial algebra. Thus every initial specification of $\mathbf{A}$ is also a weak initial specification.*

Proof. Let $\mathbf{A}$ be an initial algebra of $\mathsf{Min}\,\mathsf{K}$. Since $\mathsf{Min}\,\mathsf{K} \subseteq \mathsf{Qv}\,\mathsf{K} \subseteq \mathsf{Va}\,\mathsf{K}$, it suffices to show that $\mathbf{A}$ is also initial in $\mathsf{Va}\,\mathsf{K}$. Let $\mathbf{B}$ be an arbitrary member of $\mathsf{Va}\,\mathsf{K}$. Then by Corollary 1.3(ii), $\mathbf{B}$ is isomorphic to a subalgebra of a product quotient $(\prod_{i\in I}\mathbf{C}_i)/\Theta$, where $\mathbf{C}_i \in \mathsf{K}$ for each $i \in I$. Without loss of generality we can assume $\mathbf{B} \subseteq (\prod_{i\in I}\mathbf{C}_i)/\Theta$. Since $\mathbf{A}$ is initial in $\mathsf{Min}\,\mathsf{K}$, there exists a homomorphism $h_i : \mathbf{A} \to \mathbf{Min}\,\mathbf{C}_i \subseteq \mathbf{C}_i$ for each $i \in I$. Then $h(a) = \langle h_i(a) : i \in I\rangle$ defines a

homomorphism $h : \mathbf{A} \to \prod_{i \in I} \mathbf{C}_i$, and thus $n \circ h : \mathbf{A} \to (\prod_{i \in I} \mathbf{C}_i)/\Theta$, where n is the natural map. Since $\mathbf{A}$ is minimal, its image $(n \circ h)(\mathbf{A})$ is also minimal, and hence included in $\mathbf{B}$. So $n \circ h : \mathbf{A} \to \mathbf{B}$. □

COROLLARY 2.9. *Γ initially specifies $\mathbf{A}$ if and only if Γ weakly initially specifies $\mathbf{A}$ and $\mathbf{A} \in \mathsf{Min}(\mathsf{Mod}\,\Gamma)$.* □

The converse of the theorem fails: there exist sets of first-order, in fact even universal, sentences that do not initially specify any data type, although such a set is always a weak initial specification. For any natural number n let

$$\varphi_n = \forall x_0 \cdots \forall x_n \Big(\bigvee_{i<j\leq n} x_i \approx x_j \Big).$$

An algebra $\mathbf{A}$, of arbitrary homogeneous signature Σ, satisfies φ_n if and only if $\mathbf{A}$ contains at most n elements. Enrich Σ to Σ' by adjoining $n+1$ new constant symbols $\kappa_0, \cdots, \kappa_n$. Let $\mathbf{A}$ be the initial algebra of $\mathsf{Qv}(\mathsf{Mod}\,\varphi_n)$. For each pair of distinct constants κ_i and κ_j there exists a model $\mathbf{B}$ of φ_n such that $\kappa_i^{\mathbf{B}} \neq \kappa_j^{\mathbf{B}}$. Thus $\kappa_i^{\mathbf{A}} \neq \kappa_j^{\mathbf{A}}$ for all $i < j \leq n$. So $\mathbf{Min}\,\mathbf{A}$ contains more than n elements, and hence $\mathbf{A} \notin \mathsf{Min}(\mathsf{Mod}\,\varphi_n)$. Thus, by the corollary, φ_n cannot initially specify any data type.

There is an even stronger correspondence between K and $\mathsf{Qv}\,\mathsf{K}$ with regard to final algebra specification. The following result explains why the notion of weak final specification does not lead to anything new. Recall that for any class K, K_0 is the subclass of all nontrivial algebras.

THEOREM 2.10. *Let K be any elementary class of Σ-algebras. $(\mathsf{Min}\,\mathsf{K})_0$ has a final algebra if and only if $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$ has a final algebra; if these final algebras exist, they are isomorphic.*

Proof. Let $\mathbf{A}$ be a Σ-data type. We will show that $\mathbf{A}$ is final for $(\mathsf{Min}\,\mathsf{K})_0$ if and only if it is final for $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$. Assume first of all that $\mathbf{A}$ is final for $(\mathsf{Min}\,\mathsf{K})_0$. Let $\mathbf{B} \in (\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$. We must show $\mathbf{B} \models t \approx r$ implies $\mathbf{A} \models t \approx r$ for every ground equation $t \approx r$.

$\mathbf{B}$ is nontrivial and isomorphic to the minimal subalgebra of a reduced product $(\prod_{i \in I} \mathbf{C}_i)/\Theta_F$ with $\mathbf{C}_i \in \mathsf{K}$. We can assume that

$$\mathbf{B} = \mathbf{Min} \Big(\prod_{i \in I} \mathbf{C}_i \Big) / \Theta_F.$$

It is easy to check that $(\prod_{i \in I} \mathbf{Min}\,\mathbf{C}_i)/\Theta_F \subseteq (\prod_{i \in I} \mathbf{C}_i)/\Theta_F$, and hence $\mathbf{B} \subseteq (\prod_{i \in I} \mathbf{Min}\,\mathbf{C}_i)/\Theta_F$. Then $t^{\mathbf{B}} = [\langle t^{\mathbf{Min}\,\mathbf{C}_i} : i \in I \rangle]_{\Theta_F}$, for every ground term t. Suppose $t^{\mathbf{B}} = r^{\mathbf{B}}$. Then $J = \{\, i : t^{\mathbf{Min}\,\mathbf{C}_i} = r^{\mathbf{Min}\,\mathbf{C}_i} \,\} \in F$, by definition of Θ_F. Not every $\mathbf{Min}\,\mathbf{C}_i$ with $i \in F$ is trivial, since otherwise $(\prod_{i \in I} \mathbf{Min}\,\mathbf{C}_i)/\Theta_F$ and hence $\mathbf{B}$ would be trivial, contrary to the assumption $\mathbf{B} \in (\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$. Thus there exists at least one i such that $\mathbf{Min}\,\mathbf{C}_i \in (\mathsf{Min}\,\mathsf{K})_0$ and $t^{\mathbf{Min}\,\mathbf{C}_i} = r^{\mathbf{Min}\,\mathbf{C}_i}$. This implies $\mathbf{A} \models t \approx r$ since $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$. Thus $\mathbf{A}$ is final in $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$.

Now assume $\mathbf{A}$ is final in $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$. To show that $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$ it suffices to show $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K})_0$. By an argument similar to the one used above, we can assume without loss of generality that $\mathbf{A} \subseteq (\prod_{i \in I} \mathbf{C}_i)/\Theta_F$ with $\mathbf{C}_i \in \mathsf{K}$ and F a proper filter of I. Choose distinct elements a and b of $\mathbf{A}$. Then a is of the form $[\langle a_i : i \in I \rangle]_{\Theta_F}$ with $a_i \in C_i$ for all i; similarly $b = [\langle b_i : i \in I \rangle]_{\Theta_F}$. $a \neq b$ implies $J = \{\, i \in I : a_i = b_i \,\} \notin F$. Let U be any ultrafilter on I such that $F \subseteq U$ and $J \notin U$. Then $h : \mathbf{A} \to (\prod_{i \in I} \mathbf{C}_i)/\Theta_U$ where h is the natural homomorphism from $(\prod_{i \in I} \mathbf{C}_i)/\Theta_F$ onto $(\prod_{i \in I} \mathbf{C}_i)/\Theta_U$. By Łoś's theorem $(\prod_{i \in I} \mathbf{C}_i)/\Theta_U \in \mathsf{K}$, since

K is an elementary class. Thus $h(\mathbf{A}) \in \mathsf{Min}\,\mathsf{K}$. Now $h(a) = [\langle a_i : i \in I\rangle]_{\Theta_U}$ and $h(b) = [\langle b_i : i \in I\rangle]_{\Theta_U}$. Thus, since $\{\, i \in I : a_i \neq b_i \,\} = J \notin U$ by assumption, $h(a) \neq h(b)$. Thus $h(\mathbf{A}) \in (\mathsf{Min}\,\mathsf{K})_0$. But $(\mathsf{Min}\,\mathsf{K})_0 \subseteq (\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$, and $\mathbf{A}$ is final in $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$. Thus $\mathbf{A} \cong h(\mathbf{A})$, which gives $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K})_0$. □

Combining Theorems 2.6, 2.8, and 2.10, we have that, for any elementary class K, $(\mathsf{Min}\,\mathsf{K})_0$ is a data type if and only if $(\mathsf{Min}(\mathsf{Qv}\,\mathsf{K}))_0$ is one, and in this case the two classes coincide.

The various notions of specification can alternatively be characterized more directly in terms of the logical consequence relations $\models$ and $\vdash$; compare Meseguer and Goguen [34, Thm. 18].

THEOREM 2.11. *Let Γ be a set of first-order Σ-sentences, and let $\mathbf{A}$ be a Σ-data type.*

(i) *Γ is a weak initial specification of $\mathbf{A}$ if and only if, for every ground Σ-equation $t \approx r$,*

$$\mathbf{A} \models t \approx r \quad \textit{iff} \quad \Gamma \vdash t \approx r.$$

(ii) *Γ is an initial specification of $\mathbf{A}$ if and only if $\mathbf{A} \in \mathsf{Min}(\mathsf{Mod}\,\Gamma)$, and, for every ground Σ-equation $t \approx r$,*

$$\mathbf{A} \models t \approx r \quad \textit{implies} \quad \Gamma \vdash t \approx r.$$

(iii) *Γ is a final specification of $\mathbf{A}$ if and only if $\mathbf{A} \in (\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$, and, for every ground Σ-equation $t \approx r$, either $\mathbf{A} \not\models t \approx r$ implies $\Gamma, t \approx r \vdash \omega_\Sigma$, or, equivalently,*

$$\mathbf{A} \not\models t \approx r \quad \textit{implies} \quad \Gamma, \neg\omega_\Sigma \vdash \neg(t \approx r).$$

(iv) *Γ is a complete specification of $\mathbf{A}$ if and only if $\mathbf{A} \in (\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$, and, for every ground Σ-equation $t \approx r$, either*

$$\Gamma \vdash t \approx r \quad \textit{or} \quad \Gamma, \neg\omega_\Sigma \vdash \neg(t \approx r).$$

Proof. (i) Since any quasi variety is closed under the formation of isomorphic images, subalgebras, and Cartesian products, we can apply the equivalence of Lemma 2.2(iv) and (v), with $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ in place of K, to conclude that $\mathbf{A}$ is initial in $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ if and only if

$$\mathbf{A} \models t \approx r \quad \text{iff} \quad \mathsf{Qv}(\mathsf{Mod}\,\Gamma) \models t \approx r.$$

But by definition, $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ satisfies exactly the same conditional equations $\mathsf{Mod}\,\Gamma$ does; in particular

$$\mathsf{Qv}(\mathsf{Mod}\,\Gamma) \models t \approx r \quad \text{iff} \quad \Gamma \vdash t \approx r,$$

for each ground equation $t \approx r$. Thus (i) holds.

(ii) Under the assumption $\mathbf{A} \in \mathsf{Min}(\mathsf{Mod}\,\Gamma)$, $\Gamma \vdash t \approx r$ implies $\mathbf{A} \models t \approx r$ for every ground equation $t \approx r$. Part (ii) now follows immediately from (i) and Corollary 2.9.

(iii) Assume Γ is a final specification of $\mathbf{A}$, i.e., $\mathbf{A}$ is final in $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$. Applying Lemmas 2.3 and 2.4 we get

$$\begin{aligned} \mathbf{A} \text{ is final in } (\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0 \quad &\text{iff} \quad \mathbf{A} \text{ is final in } \mathsf{Min}(\mathsf{Mod}(\Gamma \cup \{\neg\omega_\Sigma\})) \\ &\text{iff} \quad \mathbf{A} \in (\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0 \text{ and } \mathbf{A} \not\models t \approx r \text{ implies } \Gamma, \neg\omega_\Sigma \vdash \neg(t \approx r). \end{aligned}$$

Part (iv) is immediate from (ii) and (iii). □

COROLLARY 2.12. *Let* **A** *be a* Σ*-data type.*

(i) *If* **A** *has a finite, first-order initial or weak initial specification, then it is semicomputable.*

(ii) *If* **A** *has a finite, first-order final specification, then it is cosemicomputable.*

(iii) *If* **A** *has a finite, first-order complete specification, then it is computable.*

Proof. (i) If Σ is either an initial or weak initial specification of $\mathbf{A}$, then $\mathbf{A} \models t \approx r$ if and only if $\Gamma \vdash t \approx r$ for every ground equation. If Γ is finite, then $\{t \approx r : \Gamma \vdash t \approx r\}$ is a r.e. set. This gives part (i).

(ii) Let Σ be a finite set of first-order sentences and generate a list of ground equations as follows. Recursively enumerate all first-order proofs that use sentences from $\Sigma \cup \{t \approx r\}$, for an arbitrary ground equation $t \approx r$, as nonlogical axioms. Add $t \approx r$ to the list whenever a proof of ω_Σ from $\Sigma \cup \{t \approx r\}$ appears in the enumeration. The set of ground equations generated in this way is r.e. and, by Theorem 2.11(iii), coincides with the set $\{t \approx r : \mathbf{A} \not\models t \approx r\}$ whenever Γ is a final specification of $\mathbf{A}$. This gives (ii); (iii) follows immediately from (i) and (ii). □

The corollary also holds for recursive or even r.e. specifications.

Bergstra and Tucker [1]–[3] have obtained surprising and highly significant converses of all three results in Corollary 2.12. They prove that every computable data type has a finite complete equational specification and every cosemicomputable data type has a finite final conditional equational specification; both specifications require hidden operations but not hidden sorts. They also show that every semicomputable data type has a finite equational initial specification, but in this case a hidden sort is required as well as hidden operations. For related results, see Bergstra, Broy, Tucker, and Wirsing [4]; Marongiu and Tulipani [31]; and Moss, Meseguer, and Goguen [35].

Example 2.13. As an application of part (iv) of the theorem we give a universal complete specification of stacks of natural numbers with errors. Let $S = \{\, nat, stk \,\}$ and

$$\Sigma = \{\, zero, succ, push, pop, top, empty, naterr, stkerr \,\}.$$

NATSTK is the Σ-data structure whose *nat* domain is the natural numbers with zero, the successor operation, and a special error element. The *stk* domain consists of all finite sequences of natural numbers, together with a special stack error element. The push operation takes a stack and number as argument; it returns a stack. The pop and top operations take a stack as single argument and return, respectively, a stack and number. The operations are defined in the obvious way, with push returning an error if either of its arguments is an error, and pop and push both returning an error if the argument is either the empty stack (sequence) or error.

Let Δ be the following system of universal sentences (we omit universal quantifiers for simplicity):

$\neg(zero \approx naterr)$,
$\neg(succ(x) \approx zero)$,
$succ(x) \approx succ(y) \rightarrow x \approx y$,
$succ(naterr) \approx naterr$,
$\neg(empty \approx stkerr)$,
$pop(push(s, x)) \approx s \,\vee\, x \approx naterr$,
$top(push(s, x)) \approx x \,\vee\, s \approx stkerr$,
$(s \approx stkerr \,\vee\, x \approx naterr) \;\rightarrow\; push(s, x) \approx stkerr$,
$(s \approx empty \,\vee\, s \approx stkerr) \;\rightarrow\; (pop(s) \approx stkerr \,\wedge\, top(s) \approx naterr)$.

Δ is a complete specification of **NATSTK**. Since **NATSTK** is a model of Δ, and $\Delta \vdash \neg\omega_\Sigma$, it suffices to prove that $\Delta \vdash t \approx r$ or $\Delta \vdash \neg(t \approx r)$ for every ground Σ-equation $t \approx r$. The proof is straightforward. A *nat* term t is in *canonical form* if $t = succ^n(zero)$ for some natural number n, or $t = naterr$. An *stk* term t is in canonical form if $t = push^k(empty, succ^{n_0}(zero), \cdots, succ^{n_{k-1}}(zero))$ for natural numbers $k, n_0, \cdots, n_{k-1}$ with $k > 0$, or $t = empty$, or $t = stkerr$. Using structural induction we first prove that, for every ground term t, there exists a term t' in canonical form such that $\Sigma \vdash t \approx t'$, and then that $\Delta \vdash \neg(t \approx r)$ for any pair of distinct terms in canonical form.

Lemma 2.2, on which the last theorem is partly based, can also be used to give a somewhat different perspective on Theorem 2.8. Let K be any class of Σ-algebras. Whether a given data type $\mathbf{A} \in \mathsf{Min}\,\mathsf{K}$ is initial in $\mathsf{Min}\,\mathsf{K}$ depends only on the equations, in particular the ground equations, satisfied by K. Thus $\mathbf{A}$ is initial in any one of the classes $\mathsf{Min}\,\mathsf{K}$, $\mathsf{Qv}\,\mathsf{K}$, and $\mathsf{Va}\,\mathsf{K}$ if and only if it is initial in all of them. A similar observation can be made about Theorem 2.10. Assume $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K})_0$. Lemma 2.3 can be used to show that $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$ if and only if, for every ground equation $t \approx r$,

$$\mathbf{A} \not\models t \approx r \quad \text{iff} \quad \mathsf{K} \models t \approx r \rightarrow p \approx q \qquad \text{for every } p \approx q \in \Omega_\Sigma.$$

Thus, whether or not $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$ depends only on the conditional equations satisfied by K. Hence $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$ or $(\mathsf{Qv}\,\mathsf{K})_0$ if and only if it is final in both of them. This does not, however, express the full content of Theorem 2.10 when K is an elementary class. The theorem asserts, in effect, that a data type $\mathbf{A}$ is final for $(\mathsf{Min}\,\mathsf{K})_0$ or $(\mathsf{Qv}\,\mathsf{K})_0$ if and only if it is final for both of them, without the qualification that $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K})_0$. This stronger result seems to require the model-theoretic argument using ultraproducts employed in the proof of Theorem 2.10; at least we can see no convenient way of avoiding it.

The characterizations of the initial and final specifications given in Theorem 2.11 are closely related to the work of Wirsing, Broy, and Pair [47], [48]. For each Σ-data type $\mathbf{A}$ there is a unique surjective homomorphism $h : \mathbf{T}_\Sigma \rightarrow \mathbf{A}$, and a unique congruence relation $\equiv_\mathbf{A}$ on $\mathbf{T}_\Sigma$ (the relation kernel of h), such that $\mathbf{T}/\equiv_\mathbf{A} \cong \mathbf{A}$. Conversely, each congruence on $\mathbf{T}_\Sigma$ determines a unique data type. This establishes a bijection between $Con\,\mathbf{T}_\Sigma$ and the data types of signature σ. Wirsing and Broy [47] essentially identify data types with congruences on $\mathbf{T}_\Sigma$. There is a (necessarily unique) isomorphism between data types $\mathbf{A}$ and $\mathbf{B}$ if and only if $\equiv_\mathbf{A}$ and $\equiv_\mathbf{B}$ coincide; thus any categorical property of data types can be reformulated in terms of the partial ordering on $Con\,\mathbf{T}_\Sigma$ given by set-theoretical inclusion. For any class K of Σ-algebras closed under isomorphism let

$$Con_\mathsf{K}\mathbf{T}_\Sigma = \{\,\Theta \in Con\,\mathbf{T}_\Sigma : \mathbf{T}_\Theta \in \mathsf{Min}\,\mathsf{K}\,\}.$$

Initial and final algebras of $\mathsf{Min}\,\mathsf{K}$ exist if and only if $Con_\mathsf{K}\mathbf{T}_\Sigma$ has a least upper bound and a greatest lower bound, respectively. When $\mathsf{K} = \mathsf{Mod}\,\Gamma$ for some set of first-order sentences, Wirsing and Broy [47] give sufficient conditions on the structure of the sentences of K which ensure that these bounds exist.

3. Equality-test algebras and their specification. From the discussion of the previous section we see that, when specifying a data type $\mathbf{A}$ (either initially or finally) by a set Γ of first-order sentences, Γ can always be replaced by a set of conditional equations, in fact, by any set of conditional equations that axiomatize

$\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$. The problem is that there is no effective procedure for constructing such a conditional specification from Γ in general; $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ may not be finitely axiomatizable, even when Γ is finite. However, if one of the sorts of the data type $\mathbf{A}$ is the two-element Boolean algebra, and $\mathbf{A}$ contains equality tests for each of the other sorts (i.e., $\mathbf{A}$ is an *equality-test algebra* in the sense of the following definition), and if Γ is a finite set of universal sentences, then $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ will always be finitely axiomatizable, and there is a simple algorithm for converting Γ into a set of conditional axioms for it, and hence into a finite conditional initial specification of $\mathbf{A}$. Moreover, the specification obtained in this way is necessarily complete. Actually, any universal initial, weak initial, or final specification of $\mathbf{A}$ gives rise in this way to a conditional complete specification of $\mathbf{A}$ (Theorem 3.14). An interesting consequence of this fact is that every semicomputable or cosemicomputable equality-test data type is computable (Theorem 3.18). The importance of conditional specifications was first recognized by Thatcher, Wagner, and Wright [43].

DEFINITION 3.1. (i) A signature Σ is called an **equality-test signature** if it has a sort *bool* with operation symbols $+$, $\cdot$, $-$, 0, 1, and, for each sort $s \neq bool$, an operation symbol $eq_s : s\, s \to bool$.

(ii) A Σ-algebra $\mathbf{A}$ is an **equality-test** (**ET** for short) **algebra** if Σ is an equality-test signature, $\mathbf{A}_{bool} = \mathbf{B}_2$ (the two-element Boolean algebra), and, for each $s \in S \setminus \{bool\}$,

$$eq_s^{\mathbf{A}}(a,b) = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{if } a \neq b. \end{cases}$$

The class of all equality-test Σ-algebras is denoted by ET_Σ, or simply ET when Σ is clear from context.

eq_s is called the *s-equality-test operation.* No fundamental equality-test operation for the Boolean sort is postulated; it can be defined in terms of the Boolean operations. Specifically, we define

$$eq_{bool}(x,y) = (-x+y)\cdot(-y+x).$$

Clearly $eq^{\mathbf{A}}_{bool}(a,b)$ equals one if $a=b$, and zero otherwise.

Note that an ET algebra has at least one domain with more than one element (the Boolean sort); hence every ET algebra is nontrivial. Also, every minimal subalgebra of an ET algebra is ET. A minimal ET algebra is called an *ET data structure* or *data type.* For the rest of this section Σ is assumed to be an equality-test signature except when otherwise noted.

3.1. Generalized equality-test algebras. We shall investigate in some detail the quasi variety generated by the class of all ET algebras of a fixed but arbitrary equality-test signature. Members of this quasi variety are called *generalized equality-test algebras.* It turns out that this quasi variety has two properties that make it especially useful for studying specifications of ET data types: it has a simple, finite axiomatization, and the structure of an arbitrary generalized ET algebra can be easily described in terms of ET algebras. In fact, a natural analogue of the algebraic version of the Stone representation theorem for Boolean algebras holds in which generalized and ordinary ET algebras, respectively, play the roles of arbitrary and two-element Boolean algebras.

DEFINITION 3.2. Let Σ be an equality-test signature. A Σ-algebra $\mathbf{A}$ is called a **generalized equality-test** (**GET**) **algebra** if it satisfies the following set of equations and conditional equations:

(Axget$_1$) The equational axioms for Boolean algebras (see §1), and for each $s \in S \setminus \{bool\}$:

(Axget$_2$) $eq_s(x,x) \approx 1$;

(Axget$_3$) $eq_s(x,y) \leq eq_s(y,x)$;

(Axget$_4$) $eq_s(x,y) \cdot eq_s(y,z) \leq eq_s(x,z)$;

(Axget$_5$) $eq_{s_0}(x_0,y_0) \cdots eq_{s_{n-1}}(x_{n-1},y_{n-1}) \leq eq_s\big(\sigma(x_0,\cdots,x_{n-1}),\sigma(y_0,\cdots,y_{n-1})\big)$, for each $\sigma \in \Sigma_{w,s}$ with $w = s_0 s_1 \cdots s_{n-1}$;

(Axget$_6$) $eq_s(x,y) \approx 1 \rightarrow x \approx y$.

The finite set of conditional axioms (Axget$_1$)–(Axget$_6$) is denoted by AXGET_Σ, and the quasi variety of all GET Σ-algebras, i.e., $\mathsf{Mod}(\mathrm{AXGET}_\Sigma)$, is denoted by GET_Σ.

The subscript $_\Sigma$ is normally omitted when there is no possibility of confusion. Notice that (Axget$_6$) is the only strictly conditional equational axiom; all the others are equations.

(Axget$_2$)–(Axget$_6$) continue to hold when the sort s is allowed to be *bool*. Moreover (Axget$_5$) continues to hold with $\sigma = eq_s$ for every $s \in S$. In this case (Axget$_5$) takes the form

$$eq_s(x_0,y_0) \cdot eq_s(x_1,y_1) \leq eq_{bool}\big(eq_s(x_0,x_1),\ eq_s(y_0,y_1)\big).$$

This follows from (Axget$_3$) and (Axget$_4$), since, in any Boolean algebra, $a \cdot b \leq c$ and $a \cdot c \leq b$ together imply $a \leq (-b+c)\cdot(-c+b)$.

Every ET Σ-algebra is a GET algebra. In fact the ET algebras are exactly the GET algebras $\mathbf{A}$ such that $\mathbf{A}_{bool} = \mathbf{B}_2$: it is easy to see that, if $A_{bool} = \{0^{\mathbf{A}}, 1^{\mathbf{A}}\}$ and $0^{\mathbf{A}} \neq 1^{\mathbf{A}}$, then (Axget$_2$) and (Axget$_6$) guarantee that $eq_s^{\mathbf{A}}$ is the equality-test function on A_s for each $s \in S \setminus \{bool\}$. This gives us Theorem 3.3.

THEOREM 3.3. *The class* ET_Σ *of all equality-test* Σ*-algebras is a finitely axiomatizable universal class. It is defined by the axioms* AXGET_Σ *and the single additional universal sentence*

(Axet) $\neg(0 \approx 1) \wedge \forall x(x \approx 0 \vee x \approx 1)$. □

We set $\mathrm{AXET}_\Sigma = \mathrm{AXGET}_\Sigma \cup \{(\mathrm{Axet})\}$, so that $\mathsf{ET}_\Sigma = \mathsf{Mod}(\mathrm{AXET}_\Sigma)$. The universal axiom (Axet) cannot be replaced by an equation or conditional equation.

GET contains many non-ET algebras. In fact, for every Boolean algebra $\mathbf{B}$ we can find many nonisomorphic $\mathbf{A} \in \mathsf{GET}$ such that $\mathbf{A}_{bool} = \mathbf{B}$. Not even every minimal GET algebra is ET. Let $\mathbf{A}_0$ and $\mathbf{A}_1$ be minimal ET algebras, and $t \approx r$ a ground equation such that $t^{\mathbf{A}_0} = r^{\mathbf{A}_0}$ but $t^{\mathbf{A}_1} \neq r^{\mathbf{A}_1}$; such algebras are easy to find. If $\mathbf{B} = \mathbf{A}_0 \times \mathbf{A}_1$, then

$$eq_s(t,r)^{\mathbf{B}} = \big\langle eq_s^{\mathbf{A}_0}(t^{\mathbf{A}_0}, r^{\mathbf{A}_0}),\ eq_s^{\mathbf{A}_1}(t^{\mathbf{A}_1}, r^{\mathbf{A}_1})\big\rangle = \langle 1^{\mathbf{A}_0}, 0^{\mathbf{A}_1}\rangle.$$

Thus $eq_s(t,r)^{\mathbf{B}} \notin \{\,\langle 0^{\mathbf{A}_0}, 0^{\mathbf{A}_1}\rangle, \langle 1^{\mathbf{A}_0}, 1^{\mathbf{A}_1}\rangle\,\} = \{0^{\mathbf{B}}, 1^{\mathbf{B}}\}$. So $\mathbf{Min}\,\mathbf{B}$ is not ET, but it is obviously GET. We shall show that $\mathsf{GET}_\Sigma = \mathsf{Qv}(\mathsf{ET}_\Sigma)$ (see Theorem 3.7 below).

On the other hand, we do have the following useful characterization of the Boolean part of any minimal GET algebra.

LEMMA 3.4. *Let* $\mathbf{A}$ *be a minimal* GET *algebra. Then* $\mathbf{A}_{bool}$ *is generated by* $0^{\mathbf{A}}$, $1^{\mathbf{A}}$, *and elements of the form* $eq_s^{\mathbf{A}}(a,b)$, *where* $s \in S \setminus \{bool\}$ *and* $a, b \in A_s$. *Thus every element of* $\mathbf{A}_{bool}$, *different from* $0^{\mathbf{A}}$ *and* $1^{\mathbf{A}}$, *can be written in the form*

$$(e_0 \cdots e_{n-1}) + (f_0 \cdots f_{m-1}) + \cdots + (g_0 \cdots g_{p-1}),$$

where each of the $e_i, f_j, \cdots, g_k$ *is of the form* $eq_s^{\mathbf{A}}(a,b)$ *or* $-eq_s^{\mathbf{A}}(a,b)$ *for some* $s \in S \setminus \{bool\}$ *and* $a, b \in A_s$.

Proof. Since $\mathbf{A}$ is minimal, every element of $\mathbf{A}_{bool}$ is of the form $t^{\mathbf{A}}$, where t is a ground *bool*-term. But every ground *bool*-term is either 0, 1, or a Boolean combination of terms of the form $eq_s(r,p)$ where $s \in S\setminus\{bool\}$ and r and p are ground s-terms. The representation given in the theorem is just the standard conjunctive normal-form representation of an element of a Boolean algebra in terms of a system of generators. □

As a corollary, for any nontrivial GET algebra $\mathbf{A}$ we have $\mathbf{A}_{bool} \neq \mathbf{B}_2$, i.e., $\mathbf{A} \notin \mathsf{ET}$, if and only if there exist $s \in S \setminus \{bool\}$ and $a, b \in A_s$ such that $eq^{\mathbf{A}}(a,b) \notin \{0^{\mathbf{A}}, 1^{\mathbf{A}}\}$.

Let $\mathbf{A}$ be an arbitrary GET algebra and $\Theta = \langle \Theta_s : s \in S\rangle$ a congruence on $\mathbf{A}$. The quotient $\mathbf{A}/\Theta$ is not necessarily a GET algebra since GET is not closed under homomorphism (the conditional axioms $eq_s(x,y) \approx 1 \rightarrow x \approx y$ are not in general preserved in passing to the homomorphic image). We call Θ a GET *congruence* if $\mathbf{A}/\Theta \in \mathsf{GET}$, or, equivalently, if $eq_s^{\mathbf{A}}(a,b) \equiv 1(\Theta_{bool})$ implies $a \equiv b(\Theta_s)$ for all $s \in S \setminus \{bool\}$. The set of all GET congruences on $\mathbf{A}$ is denoted by $Con_{\mathsf{GET}}\mathbf{A}$. Like the set $Con\,\mathbf{A}$ of all congruences on $\mathbf{A}$, $Con_{\mathsf{GET}}\mathbf{A}$ forms a complete lattice $\mathbf{Con}_{\mathsf{GET}}\mathbf{A}$ under sorted set-theoretical inclusion. The greatest lower bound of any system $\{\Theta_i : i \in I\}$ of GET congruences is again the sorted set-theoretical intersection $\bigcap_{i\in I}\Theta_i = \langle \bigcap_{i\in I}\Theta_{i,s} : s \in S\rangle$. ($Con_{\mathsf{GET}}\mathbf{A}$ is closed under intersection since GET is a quasi variety.) The least upper bound of $\{\Theta_i : i \in I\}$ is not generally the sorted union of the Θ_i, and differs from the least upper bound in $\mathbf{Con}\,\mathbf{A}$. ($\mathbf{Con}_{\mathsf{GET}}\mathbf{A}$ is not a sublattice of $\mathbf{Con}\,\mathbf{A}$.) But the identity and universal relations Δ_A and ∇_A are GET congruences, and hence are, respectively, the smallest and largest elements of $\mathbf{Con}_{\mathsf{GET}}\mathbf{A}$.

The correspondence theorem also holds for GET congruences: for every GET congruence Θ, the lattice $\mathbf{Con}_{\mathsf{GET}}(\mathbf{A}/\Theta)$ is isomorphic to the sublattice of $\mathbf{Con}_{\mathsf{GET}}\mathbf{A}$ of all GET congruences that include Θ.

If Θ is a GET congruence on $\mathbf{A}$, then Θ_{bool} is a congruence on the Boolean algebra $\mathbf{A}_{bool}$. Define $Fi\,\Theta = Fi\,\Theta_{bool} = [1^{\mathbf{A}}]_{\Theta_{bool}}$; Fi is a mapping from $Con_{\mathsf{GET}}\mathbf{A}$ into the set $Fil\,\mathbf{A}_{bool}$ of filters of $\mathbf{A}_{bool}$.

LEMMA 3.5. *Fi is an isomorphism between the lattices* $\mathbf{Con}_{\mathsf{GET}}\mathbf{A}$ *and* $\mathbf{Fil}\,\mathbf{A}_{bool}$ *for every* GET *algebra* $\mathbf{A}$.

Proof. To prove that a mapping between lattices is an isomorphism, we only need to show that it is an order-preserving bijection. Fi is order-preserving since $\Theta \subseteq \Phi$ implies $\Theta_{bool} \subseteq \Phi_{bool}$, which is equivalent to $Fi\,\Theta_{bool} \subseteq Fi\,\Phi_{bool}$. We show that Fi is injective. Let $s \in S\setminus\{bool\}$ and $a, b \in A_s$. Using the fact that Θ is a GET congruence (and hence preserves the conditional equations $eq_s(x,y) \approx 1 \rightarrow x \approx y$), we have $a \equiv b(\Theta_s)$ if and only if $eq_s^{\mathbf{A}}(a,b) \equiv_{\Theta_{bool}} eq_s^{\mathbf{A}}(a,a) = 1$ if and only if $eq_s^{\mathbf{A}}(a,b) \in Fi\,\Theta$. Thus Θ_s is uniquely determined by $Fi\,\Theta$ for each $s \in S\setminus\{bool\}$; since Θ_{bool} is obviously uniquely determined, so is Θ. Hence Fi is injective.

Let F be a filter of $\mathbf{A}_{bool}$. For each $s \in S \setminus \{bool\}$ define

$$(Co\,F)_s = \{\langle a,b\rangle \in A_s \times A_s : eq_s^{\mathbf{A}}(a,b) \in F\};$$

let $(Co\,F)_{bool} = \{\langle a,b\rangle : (-a+b)\cdot(-b+a) \in F\}$. We show that $Co\,F$ is a GET congruence, and that $Fi(Co\,F) = F$. For all $a, b \in A_s$, $eq_s^{\mathbf{A}}(a,a) = 1 \in F$ implies $\langle a,a\rangle \in (Co\,F)_s$; $\langle a,b\rangle \in (Co\,F)_s$ if and only if $eq_s^{\mathbf{A}}(a,b) \in F$, which implies $eq_s^{\mathbf{A}}(b,a) \in F$ and hence $\langle b,a\rangle \in (Co\,F)_s$, since $eq_s^{\mathbf{A}}(a,b) \leq eq_s^{\mathbf{A}}(b,a)$ by (Axget$_3$). Similarly, (Axget$_4$) and (Axget$_5$), applied for all $s \in S$, including $s = bool$, guarantee that $(Co\,F)_s$ is transitive and $Co\,F$ is preserved under substitution. Thus $Co\,F$ is a

congruence. To show that it is a GET congruence we must show that it preserves the conditional equations (Axget_6).

Suppose $s \in S\backslash\{bool\}$ and $eq_s^{\mathbf{A}}(a,b) \equiv 1\big((Co\,F)_{bool}\big)$. By definition of $(Co\,F)_{bool}$,

$$\big(-eq_s^{\mathbf{A}}(a,b)+1\big)\cdot\big(-1+eq_s^{\mathbf{A}}(a,b)\big) \;=\; eq_s^{\mathbf{A}}(a,b) \;\in\; F.$$

Thus $a \equiv b\big((Co\,F)_s\big)$ by definition of $(Co\,F)_s$. Hence

$$eq_s^{\mathbf{A}}(a,b) \equiv_{Co\,F} 1 \quad \text{implies} \quad a \equiv_{Co\,F} b,$$

and $Co\,F$ is a GET congruence.

Finally, $a \in Fi(Co\,F)$ if and only if $a \equiv 1\big((Co\,F)_{bool}\big)$ if and only if $(-a+1)\times(-1+a) = a \in F$. So $Fi(Co\,F) = F$. Thus Fi is surjective, and hence a lattice isomorphism. □

For a closely related result see Padawitz [40, Lem. 2.3].

Recall that an algebra is simple if and only if it has exactly two congruences. A GET algebra $\mathbf{A}$ is GET-*simple* if it has exactly two GET congruences, the identity relation Δ_A, and the universal relation ∇_A. So $\mathbf{A}$ is GET-simple if its only proper homomorphic GET image is the trivial algebra.

Any GET algebra simple in the absolute sense is also GET-simple, but the converse does not hold in general. For example, let Σ be the equality-test signature with just two sorts s and $bool$, and no operations other than the eq_s. Let $\mathbf{A}$ be any ET Σ-algebra. $\mathbf{A}$ is GET-simple by the following theorem, but $\langle\Theta_s, \Delta_{bool}\rangle$ is a congruence of $\mathbf{A}$ for every equivalence relation Θ_s on A_s.

THEOREM 3.6. *A* GET *algebra is* GET-*simple if and only if it is an* ET *algebra.*

Proof. By the lemma, a GET algebra $\mathbf{A}$ is GET-simple if and only if $\mathbf{A}_{bool}$ has exactly two filters, $\{1^{\mathbf{A}}\}$ and A_{bool}. But the only Boolean algebra with exactly two filters is $\mathbf{B}_2$. □

The next theorem is the analogue for GET algebras of the Stone representation theorem. A subclass of GET defined, relative to GET, by a set E of equations, i.e., a class of algebras of the form $\mathsf{Mod}(E \cup \mathrm{AXGET})$, is called a *relative subvariety of* GET. Every relative subvariety of GET is a quasi variety.

THEOREM 3.7. (i) *Let* K *be any relative subvariety of* GET. *Every member of* K *is subdirectly representable in* $\mathsf{K} \cap \mathsf{ET}$, *i.e.,* $\mathsf{K} = \mathbf{IP}_{\mathrm{SD}}(\mathsf{K} \cap \mathsf{ET})$.

(ii) *Every* GET *algebra is subdirectly representable in* ET, *i.e.,* $\mathsf{GET} = \mathbf{IP}_{\mathrm{SD}}(\mathsf{ET})$.

Proof. (i) Let $\mathsf{K} = \mathsf{Mod}(E \cup \mathrm{AXGET})$ for some set E of equations. Since K is defined by equations and conditional equations, it is a quasi variety, and hence $\mathbf{IP}_{\mathrm{SD}}(\mathsf{K} \cap \mathsf{ET}) \subseteq \mathbf{IP}_{\mathrm{SD}}\,\mathsf{K} = \mathsf{K}$. For the reverse inclusion consider any $\mathbf{A} \in \mathsf{K}$. Then $\mathbf{A} \in \mathsf{GET}$ since $\mathsf{K} \subseteq \mathsf{GET}$. If $\mathbf{A}$ is trivial, it is isomorphic to the empty subdirect product; so we may assume that it is nontrivial. Let P be the set of all prime filters of $\mathbf{A}_{bool}$. $\bigcap_{F\in P} F = \{1^{\mathbf{A}}\}$. Thus by the lemma, $\bigcap_{F\in P} Co\,F = \Delta_A$. Hence $\mathbf{A}$ is subdirectly representable in $\{\,\mathbf{A}/Co\,F : F \in P\,\}$. Applying the lemma again, we conclude from the fact that each F is prime that $Co\,F$ is a maximal proper GET congruence, and hence $\mathbf{A}/Co\,F$ is GET-simple, i.e., $\mathbf{A}/Co\,F \in \mathsf{ET}$. In particular, $\mathbf{A}/Co\,F \in \mathsf{GET}$, so $\mathbf{A}/Co\,F \models \mathrm{AXGET}$. We also have $\mathbf{A}/Co\,F \models E$ since $\mathbf{A} \models E$ by assumption and E is a set of equations, and hence preserved under surjective homomorphisms. So $\mathbf{A}/Co\,F \in \mathsf{Mod}(E \cup \mathrm{AXGET}) = \mathsf{K}$. This shows $\{\mathbf{A}/Co\,F : F \in P\} \subseteq \mathsf{K} \cap \mathsf{ET}$. Hence $\mathbf{A}$ is subdirectly representable in $\mathsf{K} \cap \mathsf{ET}$.

(ii) This is a special case of part (i) with E taken to be the empty set of equations. □

COROLLARY 3.8. $\mathsf{Qv}(\mathsf{ET}) = \mathsf{GET} = \mathsf{Mod}(\text{AXGET})$.

Thus AXGET is a set of conditional axioms for the conditional theory of ET, i.e., the set of all conditional equations that are true in every ET algebra.

The last theorem gives rise to a simple characterization of the initial algebra $\mathbf{In}\,\mathsf{K}$, and also the final algebra $\mathbf{Fn}\,\mathsf{K}$ when it exists, for every relative subvariety K of GET.

THEOREM 3.9. *Let* K *be a relative subvariety of* GET.

(i) $\mathbf{In}\,\mathsf{K} = \mathbf{Min}\big(\prod_{i\in I}\mathbf{A}_i\big)$, *where* $\{\,\mathbf{A}_i : i \in I\,\}$ *is any subset of* $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ *that includes every member of* $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ *up to isomorphism.*

(ii) *The following are equivalent:*

(ii′) $\mathbf{Fn}\,(\mathsf{Min}\,\mathsf{K})_0$ *exists;*

(ii″) $\mathbf{In}\,\mathsf{K} \in \mathsf{ET}$*;*

(ii‴) $\mathbf{In}\,\mathsf{K}$ *is isoinitial;*

(ii⁗) *There is only one algebra in* $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ *up to isomorphism.*

If these conditions hold, then $(\mathsf{Min}\,\mathsf{K})_0 = (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET} = \mathsf{I}\mathbf{A}$, *and* $\mathbf{Fn}\,(\mathsf{Min}\,\mathsf{K})_0 \cong \mathbf{In}\,\mathsf{K} \cong \mathbf{A}$, *where* $\mathbf{A}$ *is the unique data type* $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$.

Proof. Let $\mathsf{K} = \mathsf{Mod}(E \cup \text{AXGET})$ for some set E of equations.

(i) Let $\mathbf{B} \in \mathsf{K}$. By Theorem 3.7(i) there is a injective homomorphism $h : \mathbf{B} \to \prod_{j\in J}\mathbf{C}_j$ for some $\mathbf{C}_j \in \mathsf{K} \cap \mathsf{ET}$. Without loss of generality we can assume $\mathbf{B} \subseteq \prod_{j\in J}\mathbf{C}_j$. By the assumption on the set $\{\,\mathbf{A}_i : i \in I\,\}$, there exists for each $j \in J$ a $j' \in I$ such that $\mathbf{A}_{j'} \cong \mathbf{Min}\,\mathbf{C}_j$. Thus there is a homomorphism $f_j : \prod_{i\in I}\mathbf{A}_i \to \mathbf{C}_j$ that is obtained by composing the projection $\pi_{j'} : \prod_{i\in I}\mathbf{A}_i \to \mathbf{A}_{j'}$ with the unique injective homomorphism form $\mathbf{A}_{j'}$ into $\mathbf{C}_j$. Then, by the universal property of Cartesian products, there is a homomorphism $g : \prod_{i\in I}\mathbf{A}_i \to \prod_{j\in J}\mathbf{C}_j$. Restricting to the minimal subalgebra we get

$$g : \mathbf{Min}\Big(\prod_{i\in I}\mathbf{A}_i\Big) \to \mathbf{Min}\Big(\prod_{j\in J}\mathbf{C}_j\Big) \subseteq \mathbf{B}.$$

So $\mathbf{Min}\big(\prod_{i\in I}\mathbf{A}_i\big)$ is initial in K.

(ii) Suppose that $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0$ exists. By Theorem 3.7 it is a subdirect product of ET algebras in K. So it itself must be an ET algebra since otherwise it would have a proper homomorphic image in $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$, contradicting its finality. Thus $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0 \in (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$. For any other $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ there is a homomorphism $h : \mathbf{A} \to \mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0$. h is surjective because $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0$ is minimal, and it is injective since $\mathbf{A}$ is GET-simple. Thus $\mathbf{A} \cong \mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0$. This shows that (ii′) implies (ii⁗).

For the implication in the opposite direction suppose $\mathbf{A}$ is, up to isomorphism, the only member of $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$. Let $\mathbf{B}$ be any nontrivial minimal member of K. $\mathbf{B}$ is a subdirect product of members of $\mathsf{K} \cap \mathsf{ET}$ (Theorem 3.7), and, since it is minimal, it is a nonempty subdirect product of members of $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$. Thus $\mathbf{A}$ is a subdirect factor of $\mathbf{B}$, and the projection onto this factor gives a (necessarily unique) homomorphism of $\mathbf{B}$ onto $\mathbf{A}$. So $\mathbf{A}$ is final in $(\mathsf{Min}\,\mathsf{K})_0$, and is also the unique member of $(\mathsf{Min}\,\mathsf{K})_0$ up to isomorphism. Thus (ii′) and (ii⁗) are equivalent, and if these conditions hold, then $(\mathsf{Min}\,\mathsf{K})_0 = (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET} = \mathsf{I}\mathbf{A}$.

For each $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ there exists a surjective homomorphism $h : \mathbf{In}\,\mathsf{K} \to \mathbf{A}$. If $\mathbf{In}\,\mathsf{K}$ is an ET algebra, then h must be injective since $\mathbf{In}\,\mathsf{K}$ is GET-simple. Thus $\mathbf{In}\,\mathsf{K} \cong \mathbf{A}$ for each $\mathbf{A} \in (\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$. This shows that (ii″) implies (ii‴), and that (ii‴) implies (ii⁗). For the implication from (ii⁗) to (ii″), observe that, if $(\mathsf{Min}\,\mathsf{K}) \cap \mathsf{ET}$ contains only one algebra $\mathbf{A}$ up to isomorphism, then $\mathbf{In}\,\mathsf{K} \cong \mathbf{A} \in \mathsf{ET}$ by part (i). In this case $\mathbf{In}\,\mathsf{K} \cong \mathbf{A} \cong (\mathbf{Fn}\,\mathsf{K})_0$. □

Example 3.10. Let $S = \{nat, bool, set\}$ and

$$\Sigma = \{\, zero, succ, +, \cdot, -, 0, 1, empty, insert, isin, eq_{nat} \,\}.$$

NATSET is the data structure whose *nat* and *bool* domains are the usual ones and whose *set* domain is the set of finite sets of natural numbers. *empty* returns the empty set. The *insert* operation takes a number n and a finite set s and returns $s \cup \{n\}$. *isin* takes n and s and returns a Boolean: 1 if $n \in s$ and 0 otherwise. The *set* domain in **NATBAG** is the set of *bags* or *multisets* of natural numbers. In a bag the multiplicity of each member is taken into account; thus $insert(zero, insert(zero, empty))$ and $insert(zero, empty)$ denote the same set but different bags. Note that **NATSET** and **NATBAG** are not ET data structures because the equality-test for sets and bags, eq_{set}, is missing.

Let Δ consist of the equational axioms for Boolean algebras together with the following universal sentences:

$\neg(succ(x) \approx zero)$,
$succ(x) \approx succ(y) \to x \approx y$,
$insert(x, insert(y, s)) \approx insert(y, insert(x, s))$,
$isin(x, empty) \approx 0$,
$isin(x, insert(x, s)) \approx 1$,
$\neg(x \approx y) \to isin(x, insert(y, s)) \approx isin(x, s)$,
$eq_{nat}(x, x) \approx 1$,
$eq_{nat}(x, y) \leq eq_{nat}(y, x)$,
$eq_{nat}(x, y) \cdot eq_{nat}(y, z) \leq eq_{nat}(x, z)$,
$eq_{nat}(x, y) \leq eq_{nat}(succ(x), succ(y))$,
$eq_{nat}(x, y) \approx 1 \to x \approx y$.

The last six axioms are just the AXGET axioms for the $\{nat, bool\}$-reduct of Σ, i.e., the signature $\Sigma^- = \Sigma \setminus \{empty, insert, isin\}$. These, together with the first two axioms of Δ, constitute a universal complete specification of the natural numbers and Booleans, which is a reduct of both **NATSET** and **NATBAG**. The whole of Δ is both an initial specification of **NATBAG** and a final specification of **NATSET**. This is easily shown by using structural induction to verify conditions (ii) and (iii) of Theorem 2.11: For any nondecreasing string $n = n_1 n_2 \cdots n_k$ of natural numbers let

$$s_n = insert^k(\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_k, empty),$$

where $\mathbf{n}_i = succ^{n_i}(zero)$. From Δ we can prove that every ground *set*-term is equal to some s_n. It follows easily from this that $\mathbf{NATBAG} \models t \approx r$ implies $\Delta \vdash t \approx r$ for every ground equation $t \approx r$. Since $\mathbf{NATBAG} \in \mathsf{Min}(\mathsf{Mod}\,\Delta)$, we get that Δ is an initial specification by Theorem 2.11(ii). On the other hand, Δ is also a final specification of **NATSET** by Theorem 2.11(iii) because $\mathbf{NATSET} \not\models t \approx r$ implies $\Delta \vdash \neg(t \approx r)$. To show this, it clearly suffices to consider only ground *set*-terms t, r in canonical form. Let $n = n_1 \cdots n_k$ and $m = m_1 \cdots m_l$ be two nondecreasing strings of natural numbers. $\mathbf{NATSET} \not\models s_n \approx s_m$ if and only if some n_i is distinct from all m_j (or vice versa); in this case

$$\Delta \vdash isin(\mathbf{n}_i, s_n) \approx 1 \qquad \text{and} \qquad \Delta \vdash isin(\mathbf{m}_i, s_m) \approx 0.$$

But $\Delta \vdash \neg(1 \approx 0)$ because $\Delta \vdash \neg(eq_{nat}(succ(zero), zero) \approx 1)$ and Δ contains the axioms of Boolean algebra. Thus $\Delta \vdash \neg(s_n \approx s_m)$.

Let Σ^+ be the ET enrichment of Σ obtained by adjoining eq_{set} and let $\mathbf{NATBAG}^+$ and $\mathbf{NATSET}^+$ be the corresponding enriched ET data structures. Δ and even $\Delta \cup \mathrm{AXGET}_{\Sigma^+}$ fail to be initial specifications of $\mathbf{NATBAG}^+$ because, for example, $\Delta \cup \mathrm{AXGET}_{\Sigma^+} \not\vdash eq_{set}(s_{00}, s_0) \approx 0$. Similarly, $\Delta \cup \mathrm{AXGET}_{\Sigma^+}$ is not a final specification of $\mathbf{NATSET}^+$ because $\Delta \cup \mathrm{AXGET}_{\Sigma^+} \not\vdash \neg(eq_{set}(s_{00}, s_0) \approx 0)$.

3.2. Transforming universal specifications into conditional specifications. For each quantifier-free Σ-formula φ we define a *bool*-term φ^*, with the same variables as φ, by recursion on the structure of φ. If φ is an s-equation $t \approx r$, then $\varphi^* = eq_s(t, r)$. For the recursion step we take $(\varphi \wedge \psi)^* = \varphi^* \cdot \psi^*$, $(\varphi \vee \psi)^* = \varphi^* + \psi^*$, and $(\neg\varphi)^* = -(\varphi^*)$.

We extend the definition to universal sentences: let φ be an arbitrary universal sentence, and let

$$\forall x_0 \forall x_1 \cdots \forall x_{n-1} \varphi'(x_0, \cdots, x_{n-1})$$

be its prenex normal form; φ' is called the *quantifier-free matrix* of φ. We define φ^* to be the universally closed equation $\forall x_0 \cdots \forall x_{n-1} \varphi'^*$; by convention, the universal quantifiers are normally omitted. For any universal sentence φ we define $bt\,\varphi$ to be the equation $\varphi^* \approx 1$; $bt\,\varphi$ is called the *Boolean transform* of φ. For any set Γ of universal sentences we take $bt\,\Gamma = \{\, bt\,\varphi : \varphi \in \Gamma \,\}$.

The following easy lemma is a basic tool of the paper.

LEMMA 3.11. *Let* $\mathbf{A}$ *be an ET algebra and* φ *a universal sentence. Then* $\mathbf{A} \models \varphi$ *if and only if* $\mathbf{A} \models bt\,\varphi$.

Proof. Let $\varphi'(x_0, \cdots, x_{n-1})$ be the quantifier-free matrix of φ. We prove by induction on the structure of φ' that, for any assignment $\bar{a} = a_0, \cdots, a_{n-1}$ of elements of $\mathbf{A}$ to the variables $\bar{x} = x_0, \cdots, x_{n-1}$ of φ, we have $\mathbf{A} \models \varphi'[\bar{a}]$ if and only if $(\varphi'^*)^{\mathbf{A}}(\bar{a}) = 1$. Thus, since $\mathbf{A} \models \varphi$ if and only if $\mathbf{A} \models \varphi'[\bar{a}]$ for all $\bar{a}$, we get $\mathbf{A} \models \varphi$ if and only if the equation $bt\,\varphi$ holds in $\mathbf{A}$ for every interpretation of the variables, i.e., if and only if $bt\,\varphi$ is an identity of $\mathbf{A}$.

We consider the various possibilities for φ' in order.

$\varphi' = t(\bar{x}) \approx r(\bar{x})$, an s-equation:

$$\begin{aligned} \mathbf{A} \models \varphi'[\bar{a}] \quad &\text{iff} \quad t^{\mathbf{A}}(\bar{a}) = r^{\mathbf{A}}(\bar{a}) \\ &\text{iff} \quad eq_s^{\mathbf{A}}\big(t^{\mathbf{A}}(\bar{a}), r^{\mathbf{A}}(\bar{a})\big) = 1 \\ &\text{iff} \quad eq_s(t, r)^{\mathbf{A}}(\bar{a}) = 1 \\ &\text{iff} \quad \varphi'^{*\mathbf{A}}(\bar{a}) = 1. \end{aligned}$$

$\varphi' = \psi \wedge \vartheta$:

$$\begin{aligned} \mathbf{A} \models \varphi'[\bar{a}] \quad &\text{iff} \quad \mathbf{A} \models \psi[\bar{a}] \quad \text{and} \quad \mathbf{A} \models \vartheta[\bar{a}] \\ &\text{iff} \quad \psi^{*\mathbf{A}}(\bar{a}) = 1 \quad \text{and} \quad \vartheta^{*\mathbf{A}}(\bar{a}) = 1 \\ &\text{iff} \quad \psi^{*\mathbf{A}}(\bar{a}) \cdot \vartheta^{*\mathbf{A}}(\bar{a}) = 1 \\ &\text{iff} \quad \big(\psi^* \cdot \vartheta^*\big)^{\mathbf{A}}(\bar{a}) = 1 \\ &\text{iff} \quad \varphi'^{*\mathbf{A}}(\bar{a}) = 1. \end{aligned}$$

$\varphi' = \psi \vee \vartheta$:

$$\mathbf{A} \models \varphi'[\bar{a}] \quad \text{iff} \quad \mathbf{A} \models \psi[\bar{a}] \quad \text{or} \quad \mathbf{A} \models \vartheta[\bar{a}]$$

$$\begin{aligned}&\text{iff}\quad \psi^{*\mathbf{A}}(\bar a) = 1 \quad\text{or}\quad \vartheta^{*\mathbf{A}}(\bar a) = 1\\ &\text{iff}\quad \psi^{*\mathbf{A}}(\bar a) + \vartheta^{*\mathbf{A}}(\bar a) = 1\\ &\text{iff}\quad (\psi^* + \vartheta^*)^{\mathbf{A}}(\bar a) = 1\\ &\text{iff}\quad \varphi'^{*\mathbf{A}}(\bar a) = 1.\end{aligned}$$

$\varphi' = \neg\psi$: $\mathbf{A} \models \varphi'[\bar a]$ if and only if $\mathbf{A} \not\models \psi'[\bar a]$ if and only if $\psi^{*\mathbf{A}}(\bar a) \neq 1$ if and only if $\psi^{*\mathbf{A}}(\bar a) = 0$ if and only if $(-\psi^*)^{\mathbf{A}}(\bar a) = 1$ if and only if $\varphi'^{*\mathbf{A}}(\bar a) = 1$. □

COROLLARY 3.12. *For any set Γ of universal sentences,* $(\mathsf{Mod}\,\Gamma) \cap \mathsf{ET}_\Sigma = \mathsf{Mod}(bt\,\Gamma) \cap \mathsf{ET}_\Sigma$. □

The conclusion of Lemma 3.11 depends very much on the assumption that $\mathbf{A}$ is an ET algebra, i.e., that $\mathbf{A}_{bool} = \mathbf{B}_2$ and $eq_s^{\mathbf{A}}$ tests actual equality in A_s for each $s \in S$. For an arbitrary set Γ of universal sentences, it is not true in general either that $\mathsf{Mod}\,\Gamma = \mathsf{Mod}(bt\,\Gamma)$ or that $\mathsf{Qv}(\mathsf{Mod}\,\Gamma) = \mathsf{Mod}(bt\,\Gamma)$. Indeed, every Σ-algebra in which $\mathbf{A}_{bool}$ has only one element is a model of $bt\,\Gamma$ for any set of universal sentences Γ. We also cannot conclude from the fact that Γ is a universal initial or final specification of an ET data type $\mathbf{A}$ that $bt\,\Gamma$ is an equational initial, respectively final, specification of $\mathbf{A}$. An example is given below, but in the case of final specifications this is easily seen from the above remarks, since no model of $bt\,\Gamma$ in which $\mathbf{A}_{bool}$ is a one-element algebra can have a nontrivial data type as a homomorphic image.

We shall see that every ET data type that has a finite universal specification Γ of any kind also has a finite-conditional specification of the same kind. In order to obtain such a specification we have to adjoin the conditional axioms for GET algebras to the set of equations $bt\,\Gamma$.

From the next theorem we see that, if Γ is any set of universal sentences, then the quasi variety K generated by the ET models of Γ can be defined relative to GET by a set of identities, and thus is a relative subvariety of GET. The initial and final algebras can then be characterized by Theorem 3.9.

THEOREM 3.13. *Let Γ be an arbitrary set of universal sentences.*

$$\mathsf{Qv}(\mathsf{Mod}(\Gamma \cup \mathrm{AXET})) = \mathsf{Mod}(bt\,\Gamma \cup \mathrm{AXGET}).$$

Proof. If $\mathbf{A}$ is an ET algebra such that $\mathbf{A} \models \Gamma$, then $\mathbf{A} \models bt\,\Gamma$ by Lemma 3.11. Thus $\mathsf{Qv}(\mathsf{Mod}(\Gamma \cup \mathrm{AXET})) \subseteq \mathsf{Mod}(bt\,\Gamma \cup \mathrm{AXGET})$. Conversely, assume $\mathbf{A} \models bt\,\Gamma \cup \mathrm{AXGET}$. Since $\mathbf{A}$ is a GET algebra, it is subdirectly representable in ET by Theorem 3.7. Let $\mathbf{A} \subseteq_{\mathrm{SD}} \prod_{i\in I} \mathbf{B}_i$, with $\mathbf{B}_i \in \mathsf{ET}$. Each $\mathbf{B}_i$ is a homomorphic image of $\mathbf{A}$, and hence $\mathbf{B}_i \models bt\,\Gamma$, since the members of $bt\,\Gamma$ are preserved under surjective homomorphisms. Therefore, $\mathbf{B}_i \models \Gamma$ by Lemma 3.11. So $\mathbf{B}_i \in \mathsf{Mod}(\Gamma \cap \mathrm{AXET})$ for each i, and hence $\mathbf{A} \in \mathsf{Qv}(\mathsf{Mod}(\Gamma \cup \mathrm{AXET}))$. □

Combining this theorem with Theorem 3.9 we finally get the main result of the section.

THEOREM 3.14. *Let Γ be any set of universal sentences, and $\mathbf{A}$ an ET data type. If Γ is either an initial, a weak initial, or a final specification of $\mathbf{A}$, then $bt\,\Gamma \cup \mathrm{AXGET}$ is a conditional complete specification of $\mathbf{A}$.*

Proof. Assume that Γ is a weak initial specification of $\mathbf{A}$, i.e., that $\mathbf{A}$ is initial in $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$. Then $\mathbf{A}$ is also initial in

$$\mathsf{K} = \mathsf{Mod}(bt\,\Gamma \cup \mathrm{AXGET}) = \mathsf{Qv}(\mathsf{Mod}(\Gamma \cup \mathrm{AXET})),$$

since this class contains $\mathbf{A}$ and is included in $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$. $\mathbf{A}$ is also final in $(\mathsf{Min}\,\mathsf{K})_0$ by Theorem 3.9(ii). So $bt\,\Gamma \cup \mathrm{AXGET}$ is a complete specification of $\mathbf{A}$.

Assume now that Γ is a final specification of $\mathbf{A}$, i.e., $\mathbf{A}$ is final in $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$. Then $\mathbf{A}$ is final in $\big(\mathsf{Min}(\mathsf{Qv}(\mathsf{Mod}\,\Gamma))\big)_0$ by Theorem 2.10, and hence also in

$$\big(\mathsf{Min}(\mathsf{Qv}(\mathsf{Mod}(\Gamma \cup \text{AXET})))\big)_0 = \big(\mathsf{Min}\,\mathsf{K}\big)_0.$$

Thus $\mathbf{A}$ is also initial in K by Theorem 3.9(ii), and hence $bt\,\Gamma \cup \text{AXGET}$ is a complete specification of $\mathbf{A}$. □

Every ET data type that has a finite universal initial specification also has a finite-conditional complete specification. (AXGET is finite because of our assumption that every signature is finite.)

It is easy to see from the above proof that, if Γ is any (not necessarily universal) initial or final specification of an ET data type $\mathbf{A}$, then $\Gamma \cup \text{AXGET}$ is a complete (but of course not generally a conditional) specification of $\mathbf{A}$. The adjunction of AXGET is essential here. It is easy to find universal initial specifications of an ET data type that are not complete, and similarly for final specifications. Just take Γ, respectively, to be the set of all ground equations that hold in $\mathbf{A}$ and the set of logical negations of all ground equations that fail to hold in $\mathbf{A}$. In the first case we get an initial specification that is not final provided $\mathbf{A}$ is not simple (in the absolute sense). In the second case we get a final specification that is not initial provided $\mathbf{A}$ is not isomorphic to the term algebra.

A more natural example of an initial but incomplete specification of an ET data type is the following specification of the natural numbers and Booleans.

Example 3.15. The sort set is $\{nat, bool\}$ and the signature is $\{\, succ, zero, +, \cdot, -, 0, 1\,\}$. The initial specification is given by the equational axioms of Boolean algebra and

$eq_{nat}(x,x) \approx 1,$
$eq_{nat}(x,y) \approx eq_{nat}(y,x),$
$eq_{nat}\big(succ(x), zero\big) \approx 0,$
$eq_{nat}\big(succ(x), succ(y)\big) \approx eq_{nat}(x,y).$

This is not a complete specification since it has a model in which the natural numbers are as usual but the 0 and 1 in the Boolean domain are identified. □

Of course when we adjoin AXGET to these axioms we get a complete specification. But observe that any minimal model of the axioms that is not isomorphic to the initial algebra must have $0 = 1$. So we can also get a complete specification by adjoining only the single conditional equation $eq_{nat}(x,y) \approx 1 \rightarrow x \approx y$. This is an example of general phenomenon that leads to another version of Theorem 3.14 for initial specifications.

THEOREM 3.16. *Let Γ be any set of first-order sentences and let $\mathbf{A}$ be an ET data type. If Γ is an initial specification of $\mathbf{A}$, then Γ, together with the conditional equations*

$$(*) \qquad eq_s(x,y) \approx 1 \rightarrow x \approx y \quad \text{for all } s \in S,$$

is a complete specification of $\mathbf{A}$.

Proof. It suffices to show that any proper homomorphic image of $\mathbf{A}$ that satisfies these conditional equations must be trivial. Let $\mathbf{B}$ be such an image. There is a sort s and ground s-terms t, r such that $t^{\mathbf{A}} \neq r^{\mathbf{A}}$ while $t^{\mathbf{B}} = r^{\mathbf{B}}$. From the inequality and the fact that Γ is an initial specification of $\mathbf{A}$ we get $\Gamma \vdash eq_s(t,r) \approx 0$ and hence $eq_s(t,r)^{\mathbf{B}} = 0$. The equality $eq_s(x,x) \approx 1$ is an identity of $\mathbf{A}$, since it is an ET data type, and hence it is also an identity of its homomorphic image $\mathbf{B}$. Thus from $t^{\mathbf{B}} = r^{\mathbf{B}}$ we get $eq_s(t,r)^{\mathbf{B}} = 1$. So $1 = 0$ in $\mathbf{B}$ and hence $\mathbf{B}$ is trivial because of the conditional equations $(*)$. □

Results similar to Theorems 3.14 and 3.16 have been obtained in the context of logic programming. See Hsiang and Dershowitz [21] and Paul [36], [37].

COROLLARY 3.17. *Every* ET *data type that has a finite universal weak initial, initial, or final specification is computable.*

Proof. The proof is obtained using Theorem 3.14 and Corollary 2.12. □

More generally, every ET data type with a r.e. universal weak initial specification is computable.

Every semicomputable data type **A** has a r.e. equational initial specification; just take Γ to be the set of all ground equations that hold in **A**. By 3.14 $bt\,\Gamma \cup \mathrm{AXGET}$ is a r.e. complete specification of **A**, and thus by Corollary 2.12 and the remark following it **A** is actually computable. A similar argument applies to any co-r.e. data type **A**. Let Γ be the set of sentences of the form $\neg(t \approx r)$ for all ground equations $t \approx r$ that fail to hold in **A**. Then Γ is a r.e. final specification of **A**. Again, from Corollary 2.12 and Theorem 3.14 we conclude that **A** is computable. This gives the following theorem.

THEOREM 3.18. *Every semicomputable or cosemicomputable* ET *data type is computable.* □

This theorem also has a very simple direct proof. If the set of ground equations that hold in an ET data type **A** is r.e., so is the set of ground equations that fail to hold since $\mathbf{A} \not\models t \approx r$ if and only if $\mathbf{A} \models eq_s(t,r) \approx 0$. Similarly, if the ground equations that fail to hold are r.e., then so are the ones that hold since $\mathbf{A} \models t \approx r$ if and only if $\mathbf{A} \not\models eq_s(t,r) \approx 0$.

4. Conditional specifications with hidden sorts and operations. We extend the results of the last section to arbitrary data types that are not assumed to be equality-test algebras. Every data type **A** of an arbitrary signature Σ can be enriched to an ET algebra $\mathbf{A}^+$ with enriched equality-test signature Σ^+. This is done in the obvious way by adding the two-element Boolean algebra $\mathbf{B}_2$ as a new, hidden sort domain, and then adjoining all the equality tests as hidden operations. Given a universal initial specification Γ of **A** in the signature Σ, we can form the set $bt^+\Gamma$ of Boolean transforms of Γ in the enriched signature Σ^+. In general, $bt^+\Gamma$, together with the conditional axioms AXGET^+ for the GET Σ^+-algebras, will be an initial specification of $\mathbf{A}^+$ only when Γ is a complete specification of **A** to start with; see Theorem 4.7. But **A** will always be isomorphic to the Σ-reduct of the initial algebra of $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$ (Theorem 4.6). This gives us a general method for converting a finite universal initial specification of an arbitrary data type into a finite-conditional initial specification with one hidden sort and one hidden operation for each visible sort.

The key to obtaining specifications with hidden sort and operations is the following simple lemma relating the reduct of the initial and final algebras of an arbitrary class K of algebras to the initial and final algebras of the class of visible reducts of members of K.

LEMMA 4.1. *Let Σ be any signature and Σ' an arbitrary enrichment of Σ. Let K be any class of Σ'-algebras.*

(i) *If* $\mathbf{In}\,\mathsf{K}$ *exists and* $(\mathbf{In}\,\mathsf{K})|_\Sigma$ *is minimal, then* $\mathbf{In}(\mathsf{K}|_\Sigma)$ *exists and* $\mathbf{In}(\mathsf{K}|_\Sigma) \cong (\mathbf{In}\,\mathsf{K})|_\Sigma$.

(ii) *If* $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})$ *exists, then so does* $\mathbf{Fn}(\mathsf{Min}(\mathsf{K}|_\Sigma))$ *and*

$$\mathbf{Fn}(\mathsf{Min}(\mathsf{K}|_\Sigma)) \cong \mathbf{Min}(\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})|_\Sigma).$$

Proof. (i) Let $\mathbf{A} = (\mathbf{In}\,\mathsf{K})|_\Sigma$. For each $\mathbf{B} \in \mathsf{K}|_\Sigma$ choose $\mathbf{B}' \in \mathsf{K}$ such that $\mathbf{B}'|_\Sigma = \mathbf{B}$ (in general, $\mathbf{B}'$ is not unique). Let h be the unique homomorphism $h : \mathbf{In}\,\mathsf{K} \to \mathbf{B}'$. Then $h|A : \mathbf{A} \to \mathbf{B}$, and $h|A$ is unique since $\mathbf{A}$ is minimal by hypothesis.

(ii) Let $\mathbf{A} = (\mathbf{Fn}(\mathsf{Min}\,\mathsf{K}))|_\Sigma$. For each $\mathbf{B} \in \mathsf{Min}(\mathsf{K}|_\Sigma)$ choose $\mathbf{B}' \in \mathsf{K}$ such that $\mathbf{Min}(\mathbf{B}'|_\Sigma) = \mathbf{B}$; we may assume $\mathbf{B}'$ is minimal. Let $h : \mathbf{B}' \to \mathbf{Fn}(\mathsf{Min}\,\mathsf{K})$. Then $h|B : \mathbf{B} \to \mathbf{A}$, and since $\mathbf{B}$ is minimal, so is its image $h(\mathbf{B})$. Thus $h|B : \mathbf{B} \to \mathbf{Min}\,\mathbf{A}$; $h|B$ is unique since $\mathbf{B}$ is minimal. □

DEFINITION 4.2. Let Σ be an arbitrary signature with sort set S. We define an enrichment Σ^+ of Σ as follows. The enriched sort set is $S \cup \{newbool\}$. $\Sigma^+_{ss,newbool} = \{eq_s\}$ for each $s \in S$, $\Sigma^+_{newbool^2,newbool} = \{+,\cdot\}$, $\Sigma^+_{newbool,newbool} = \{-\}$, and $\Sigma^+_{\lambda,newbool} = \{0,1\}$; $\Sigma^+_{w,s} = \Sigma_{w,s}$ in all other cases. Σ^+ is called the **equality-test enrichment** of Σ. *newbool* is called the **hidden sort** and the eq_s the **hidden operations** of Σ^+. The sorts and operations of Σ are called the **visible sorts** and **visible operations** of Σ^+. Let $\mathbf{A}$ be an arbitrary Σ-algebra. The **ET enrichment** $\mathbf{A}^+$ of $\mathbf{A}$ is the Σ^+-algebra defined as follows: $\mathbf{A}^+|_\Sigma = \mathbf{A}$, $\mathbf{A}^+_{newbool} = \mathbf{B}_2$, and $eq_s^{\mathbf{A}^+}$ is the equality test operation on the domain A_s for each $s \in S$.

The notion of ET enrichment is closely related to that of *equality enrichment* considered in Goguen [13] (see also Meseguer and Goguen [34, §6.7]). However, the latter notion is proof-theoretic in character rather than model-theoretic.

If Σ already has Boolean sort *bool* we allow for the possibility that *newbool* and *bool* are the same; this was the case in Example 3.10. $\mathbf{A}^+$ is always nontrivial since its Boolean part contains two distinct elements. Throughout this section we distinguish those notions that are defined relative to the enriched type Σ^+ by using the superscript $^+$. For example, $\mathrm{AXGET}_{\Sigma^+}$ and GET_{Σ^+} are denoted, respectively, by AXGET^+ and GET^+. Also, for any set Γ of Σ-sentences, the set of Boolean transforms of Γ in the language of Σ^+ is written $bt^+\Gamma$.

The following result, which is the main lemma of the section, connects the models of a set Γ of universal Σ-sentences with the members of the relative subvariety of GET^+ defined by the equations $bt^+\Gamma$.

THEOREM 4.3. *For any signature Σ and any set Γ of universal Σ-sentences,*

$$\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+) \cap \mathsf{ET}^+ = \{\, \mathbf{A}^+ : \mathbf{A} \in \mathsf{Mod}\,\Gamma \,\}.$$

Proof. If $\mathbf{A} \models \Gamma$, then $\mathbf{A}^+ \models \Gamma$ since Γ does not involve any of the hidden operations. Hence $\mathbf{A}^+ \models bt^+\Gamma$ by Lemma 3.11, and so $\mathbf{A}^+ \in \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+) \cap \mathsf{ET}^+$. For the inclusion in the reverse direction, consider $\mathbf{B} \in \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+) \cap \mathsf{ET}^+$. $\mathbf{B} \models bt^+\Gamma$, and hence $\mathbf{B} \models \Gamma$ by Lemma 3.11. So $\mathbf{B}|_\Sigma \in \mathsf{Mod}\,\Gamma$, and $\mathbf{B} = (\mathbf{B}|_\Sigma)^+$. □

Note that the mapping $\mathbf{A} \mapsto \mathbf{A}^+$ is actually a bijection between $\mathsf{Mod}\,\Gamma$ and $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+) \cap \mathsf{ET}^+$. $\mathbf{A}^+$ is an ET algebra and hence nontrivial even if $\mathbf{A}$ is trivial. But in the latter case $A_{newbool}$ is the only nontrivial domain and 0 is not in the range of any equality-test operation (other than $eq_{newbool}$). So $\mathbf{A}^+$ is nontrivial only in a technical sense. This causes some problems, such as in Theorem 4.5 below.

One consequence of Theorem 4.3 is a simple characterization of the initial and final algebras of $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$ in terms of the models of Γ.

THEOREM 4.4. *Let Σ be any signature and Γ any set of universal Σ-sentences; let $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$.*

(i) $\mathbf{In}\,\mathsf{K} = \mathbf{Min}\big(\prod_{i\in I} \mathbf{A}_i^+\big)$, *where $\{\, \mathbf{A}_i : i \in I \,\}$ is any subset of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ that includes every member of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ up to isomorphism.*

(ii) *The following are equivalent:*

(ii′) $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0$ *exists;*

(ii″) $\mathbf{In}\,\mathsf{K} \in \mathsf{ET}^+$*;*

(ii‴) $\mathbf{In}\,\mathsf{K}$ *is isoinitial;*

(ii⁗) *there is only one algebra in* $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ *up to isomorphism.*

If these conditions hold, then $\mathbf{Fn}(\mathsf{Min}\,\mathsf{K})_0 \cong \mathbf{In}\,\mathsf{K} \cong \mathbf{A}^+$ *where* $\mathbf{A}$ *is the unique member of* $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$.

Proof. Take, respectively, Σ^+ and $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$ for Σ and K in Theorem 3.9, and use the fact that $(\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)) \cap \mathsf{ET}^+ = \{\mathbf{A}^+ : \mathbf{A} \in \mathsf{Mod}\,\Gamma\}$. □

Another consequence of Theorem 4.3 is a characterization of the quasi variety generated by $\mathsf{Mod}\,\Gamma$. Part (ii) of the following result is an analogue of Theorem 3.13.

THEOREM 4.5. *Let* Σ *be any signature, and* Γ *an arbitrary set of universal* Σ*-sentences.*

(i) $\mathsf{Mod}\,\Gamma = \big(\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+) \cap \mathsf{ET}^+\big)|_\Sigma$*;*

(ii) $\mathsf{Qv}(\mathsf{Mod}\,\Gamma) = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)|_\Sigma$.

Proof. Part (i) is a trivial corollary of Theorem 4.3.

(ii) Let $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$. From part (i) we get $\mathbf{SP}_{\mathrm{R}}(\mathsf{Mod}\,\Gamma) \subseteq \mathbf{SP}_{\mathrm{R}}(\mathsf{K}|_\Sigma) = \mathbf{S}((\mathbf{P}_{\mathrm{R}}\,\mathsf{K})|_\Sigma)$; the last equality holds because for any enrichment Σ' of Σ, any system $\langle \mathbf{A}_i : i \in I\rangle$ of Σ'-algebras, and any filter F on I, it is easy to check that

$$\Big(\Big(\prod_{i\in I}\mathbf{A}_i\Big)\Big/\Theta_F\Big)\Big|_\Sigma = \Big(\prod_{i\in I}\mathbf{A}_i|_\Sigma\Big)\Big/\Theta_F.$$

In general the formation of subalgebras does not commute with the formation of reducts, but it does in the present situation since none of the hidden operations of Σ^+ has a visible sort, i.e., a sort of Σ, as target. Thus

$$\mathbf{SP}_{\mathrm{R}}(\mathsf{Mod}\,\Gamma) \subseteq (\mathbf{SP}_{\mathrm{R}}\,\mathsf{K})|_\Sigma = \mathsf{K}\,|_\Sigma.$$

This gives one of the two inclusions of (ii).

For the reverse inclusion, consider any $\mathbf{B} \in \mathsf{K}$. We can use Theorem 3.7(i) with Σ^+ in place of Σ. $\mathbf{B}$ is subdirectly representable in $\mathsf{K} \cap \mathsf{ET}^+$. So $\mathbf{B} \subseteq_{\mathrm{SD}} \prod_{i\in I}\mathbf{C}_i$, with $\mathbf{C}_i \in \mathsf{K} \cap \mathsf{ET}^+$. Thus

$$\mathbf{B}\,|_\Sigma \subseteq_{\mathrm{SD}} \Big(\prod_{i\in I}\mathbf{C}_i\Big)\Big|_\Sigma = \prod_{i\in I}(\mathbf{C}_i|_\Sigma).$$

But $\mathbf{C}_i|_\Sigma \in \mathsf{Mod}\,\Gamma$ by part (i). Hence $\mathbf{B}\,|_\Sigma \in \mathbf{SP}_{\mathrm{R}}(\mathsf{Mod}\,\Gamma) \subseteq \mathsf{Qv}(\mathsf{Mod}\,\Gamma)$. □

The next two theorems are the main results of the section.

THEOREM 4.6. *Let* Σ *be any signature,* Γ *any set of universal* Σ*-sentences, and* $\mathbf{A}$ *a* Σ*-data type. If* Γ *is an initial specification of* $\mathbf{A}$*, or, more generally, a weak initial specification of* $\mathbf{A}$*, then* $bt^+\Gamma \cup \mathrm{AXGET}^+$ *is a conditional initial specification of* $\mathbf{A}$ *with hidden sort and operations, i.e.,* $\mathbf{A} \cong (\mathbf{In}\,\mathsf{K})|_\Sigma$*, where* $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$.

Proof. This is an immediate consequence of the second part of Theorem 4.5, together with Lemma 4.1. $(\mathbf{In}\,\mathsf{K})|_\Sigma$ is a minimal Σ-algebra since $\mathbf{In}\,\mathsf{K}$ is a minimal Σ^+-algebra, and none of the hidden operations have a visible sort as target. Thus $(\mathbf{In}\,\mathsf{K})|_\Sigma$ is initial in $\mathsf{Qv}(\mathsf{Mod}\,\Gamma)$ by Lemma 4.1(i) and Theorem 4.5(ii). □

Every data type with a finite universal initial specification has a finite-conditional initial specification with hidden sort and operations. ET enrichments are always *sufficiently complete* in the sense of Guttag and Horning [19], [20].

It is interesting to compare this result with Theorem 3.14. When an initial specification of an ET data type is relativized to the class of GET algebras by adjoining the GET axioms, we always obtain a complete specification by Theorem 3.14. However, when an initial specification of an arbitrary data type is relativized to GET^+ algebras in the same way, one does not ordinarily obtain a complete specification with hidden sort and operations. Indeed, by the next theorem, this occurs just in case the original specification is itself complete, provided it has no trivial models.

THEOREM 4.7. *Let Σ be any signature. Let $\mathbf{A}$ be a Σ-data type and let Γ be a universal initial specification of $\mathbf{A}$ such that Γ has no trivial models. The following are equivalent:*

(i) $bt^+\Gamma \cup \text{AXGET}^+$ *is a conditional complete specification of* $\mathbf{A}$ *with hidden sort and operations;*

(ii) $bt^+\Gamma \cup \text{AXGET}^+$ *is a conditional initial specification of* $\mathbf{A}^+$*;*

(iii) Γ *is a complete specification of* $\mathbf{A}$.

Proof. The equivalence of (i) and (ii) follows easily from Theorem 3.14. We prove that (ii) and (iii) are equivalent.

Let $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \text{AXGET}^+)$. (ii) is equivalent to the condition $\mathbf{In}\,\mathsf{K} \cong \mathbf{A}^+$, which in turn is equivalent to $(\mathbf{In}\,\mathsf{K})|_\Sigma \cong \mathbf{A}$ and $\mathbf{In}\,\mathsf{K} \in \mathsf{ET}^+$. Under the assumption that Γ has no trivial models, $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0 = \mathsf{Min}(\mathsf{Mod}\,\Gamma)$. Thus, by Theorem 4.4(ii), $\mathbf{In}\,\mathsf{K} \in \mathsf{ET}^+$ if and only if $(\mathsf{Min}(\mathsf{Mod}\,\Gamma))_0$ contains only one algebra up to isomorphism. But this is exactly the condition for Γ to be a complete specification of $(\mathbf{In}\,\mathsf{K})|_\Sigma$ (Theorem 2.6). □

Any data type $\mathbf{A}$ to which Theorem 4.7 applies is necessarily computable, provided only that Γ is r.e. A result closely related to Theorem 4.7 but formulated exclusively in terms of computable data types can be found in Meseguer and Goguen [34, Thm. 71].

Example 2.13 (continued). Let Δ be the universal complete specification of the data type **NATSTK** of stacks of natural numbers that was given in §2. The set $bt^+\,\Delta$ of Boolean transforms is equivalent to the following equations (relative to the axioms of Boolean algebras):

$eq_{nat}(zero, naterr) \approx 0,$

$eq_{nat}\big(succ(x), zero\big) \approx 0,$

$eq_{nat}\big(succ(x), succ(y)\big) \le eq_{nat}(x, y),$

$succ(naterr) \approx naterr,$

$eq_{stk}(empty, stkerr) \approx 0,$

$eq_{stk}\big(pop(push(s,x)),\, s\big) + eq_{stk}(x, naterr) \approx 1,$

$eq_{stk}\big(top(push(s,x)),\, x\big) + eq_{stk}(s, stkerr) \approx 1,$

$eq_{stk}(s, stkerr) + eq_{nat}(x, naterr) \le eq_{stk}\big(push(s,x),\, stkerr\big),$

$eq_{stk}(s, empty) + eq_{stk}(s, stkerr) \le eq_{stk}\big(pop(s),\, stkerr\big) \cdot eq_{nat}\big(top(s),\, naterr\big).$

For example, the Boolean transform of the third formula of Δ, the conditional equation $succ(x) \approx succ(y) \to x \approx y$, is equivalent to the Boolean inclusion

$$eq_{nat}\big(succ(x), succ(y)\big) \le eq_{nat}(x, y),$$

because

$$\begin{aligned}\big(succ(x) \approx succ(y) \to x \approx y\big)^* &= \big(\neg(succ(x) \approx succ(y)) \vee x \approx y\big)^* \\ &= -eq_{nat}\big(succ(x), succ(y)\big) + eq_{nat}(x, y).\end{aligned}$$

By Theorem 4.7, these equations, together with AXGET^+, form an initial specification of the ET enrichment of **NATSTK**, and hence a conditional complete specification of **NATSTK** with hidden sort and operations.

Example 3.10 (continued). Δ is an initial specification of $\mathbf{NATBAG}^+$ but it is not complete. Thus by Theorem 4.7 $bt^+\Delta \cup \mathrm{AXGET}^+$ cannot be an initial specification of **NATBAG**. In Example 3.10 this is shown directly. (Note that $\mathrm{AXGET}^+ = \mathrm{AXGET}_{\Sigma^+}$ and $bt^+\Delta \cup \mathrm{AXGET}^+$ is logically equivalent to $\Delta \cup \mathrm{AXGET}_{\Sigma^+}$.)

Thatcher, Wagner, and Wright [44] show that conditional specification is more powerful than equational specification by giving a simple example of a computable data type with a finite-conditional initial specification but no finite-equational one. As an application of Theorem 4.6 we construct an example of a GET algebra with this property. The construction depends on the existence of a finite (homogeneous) algebra whose identities are not finitely based (i.e., not finitely axiomatizable). Many algebras with this property are known; see, for instance, Tarski [42].

Example 4.8. Let $\mathbf{A} = \langle A, * \rangle$ be a finite *groupoid*, i.e., a homogeneous algebra with a single, binary operation $*$. The class ISA of all isomorphic images of subalgebras of $\mathbf{A}$ is a universal class by Theorem 1.2, and hence can be defined by a finite set Δ of universal sentences. We want to enrich the signature of groupoids to one that includes an infinite set of constants of the underlying groupoid sort. In order to do this and still keep the enriched signature finite we first adjoin the natural numbers as a new sort, and then a function that maps the natural numbers into the groupoid domain. Let $S = \{\, grp, nat \,\}$ and $\Sigma = \{\, *, zero, succ, gen \,\}$, where $* : grp\, grp \to grp$, $zero : \lambda \to nat$, $succ : nat \to nat$, and $gen : nat \to grp$. Let $g_n = gen(succ^n)$ for each natural number n. Observe that every ground Σ-equation of sort grp is obtained from a pure groupoid equation $t(x_0, x_1, \cdots, x_{n-1}) \approx r(x_0, x_1, \cdots, x_{n-1})$ by replacing the variable symbols by constant terms $g_0, \cdots, g_{n-1}$. Finally, let $\mathsf{K} = \mathsf{Mod}(bt^+\Delta \cup \mathrm{AXGET}^+)$. The initial algebra $\mathbf{In}\,\mathsf{K}$ of K has a finite-conditional specification $bt^+\Delta \cup \mathrm{AXGET}^+$. $(\mathbf{In}\,\mathsf{K})|_\Sigma$ is initial in $\mathsf{K}|_\Sigma$ by Lemma 4.1, and $\mathsf{K}|_\Sigma = \mathsf{Qv}(\mathsf{Mod}\,\Delta) = \mathsf{Qv}(\mathsf{ISA}) = \mathsf{Qv}\mathbf{A}$ by Theorem 4.5(ii). Hence a ground Σ^+-equation $t(g_0, g_1, \cdots, g_{n-1}) \approx r(g_0, g_1, \cdots, g_{n-1})$ is an identity of $\mathbf{In}\,\mathsf{K}$ if and only if the corresponding pure groupoid equation $t(x_0, x_1, \cdots, x_{n-1}) \approx r(x_0, x_1, \cdots, x_{n-1})$ is an identity of $\mathbf{A}$. Thus if we take $\mathbf{A}$ to be a finite groupoid whose identities are not finitely axiomatizable, we get a data type $\mathbf{In}\,\mathsf{K}$ with a finite-conditional initial specification but no finite-equational one.

We do not know of any example of an ET algebra with this property. It is also open if there exists a data type with a finite universal initial specification, but no finite-conditional one; the problem is also open for weak initial specifications. If $\mathbf{A}$ is a finite groupoid whose identities are not consequences of a finite subset of the quasi identities of the infinitely generated free algebra of $\mathsf{Qv}\mathbf{A}$, then the same data type $\mathbf{In}\,\mathsf{K}$ constructed above would provide a counterexample in the weak initial case. But it is open if such a groupoid exists. In a personal communication, Wroński has informed me that he has been able to verify the failure of this property for many of the familiar examples of finite groupoids whose identities are known not to be finitely axiomatizable (see [38]).

The following companion of Theorem 3.18 is a consequence of Theorem 4.7.

THEOREM 4.9. *Let* $\mathbf{A}$ *be a* Σ*-data type for an arbitrary signature.* $\mathbf{A}^+$ *is computable* (*or semicomputable or cosemicomputable*) *if and only if* $\mathbf{A}$ *is computable.*

Proof. Clearly, if $\mathbf{A}^+$ is computable, so is $\mathbf{A}$. Assume conversely that $\mathbf{A}$ is computable. Then $\mathbf{A}$ has a recursive universal complete specification Γ. (We can

take Γ to be the union of the set of all ground equations that hold in $\mathbf{A}$ and the set of all logical negations of ground equations that fail to hold.) By Theorem 4.7 $bt^+\Gamma \cup \mathrm{AXGET}^+$ is a recursive conditional complete specification of $\mathbf{A}^+$. So $\mathbf{A}$ is computable by Corollary 2.12. □

Like Theorem 3.18, this theorem also has a simple direct proof.

4.1. Encoding the universal theory of a data type. For any set Γ of universal Σ-sentences, the Boolean part of the initial algebra of $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$ encodes the entire universal theory of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ within its equational theory. As a consequence the complexity of the structure of the Boolean part of this initial algebra is in a sense a measure of the degree of completeness of the specification. Under the condition that it has no trivial models, Γ is complete if and only if the Boolean part of the initial algebra is the two-element Boolean algebra.

Recall that for a Σ-sentence φ we write $\Gamma \models_{\mathrm{Min}} \varphi$ to mean that $\mathbf{A} \models \varphi$ for every $\mathbf{A} \in \mathsf{Min}(\mathsf{Mod}\,\Gamma)$. Also recall that the Boolean transform $bt^+\varphi$ of φ is a Σ^+-equation of sort *newbool* of the form $\varphi^* \approx 1$.

THEOREM 4.10. *Assume that Σ is an arbitrary signature and Γ is any set of universal Σ-sentences. Then, for any universal Σ-sentence φ,*

$$\Gamma \models_{\mathrm{Min}} \varphi \quad \textit{iff} \quad \mathbf{In}\,\mathsf{K} \models bt^+\varphi,$$

where $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$.

Proof. By Theorem 4.4(i), $\mathbf{In}\,\mathsf{K} = \mathsf{Min}\big(\prod_{i\in I} \mathbf{A}_i^+\big)$, where $\{\,\mathbf{A}_i : i \in I\,\}$ is any subset of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ that includes every member of $\mathsf{Min}(\mathsf{Mod}\,\Gamma)$ up to isomorphism. Suppose $\Gamma \models_{\mathrm{Min}} \varphi$. Then $\mathbf{A}_i^+ \models \varphi$, and hence, by Lemma 3.11, $\mathbf{A}_i^+ \models bt^+\varphi$, for all $i \in I$. So $\mathbf{In}\,\mathsf{K} \models bt^+\varphi$. Conversely, if $\Gamma \not\models_{\mathrm{Min}} \varphi$, then $\mathbf{A}_i^+ \not\models bt^+\varphi$ for some i, and hence $\mathbf{In}\,\mathsf{K} \not\models bt^+\varphi$. □

As a special case of this result, we take φ to be either a ground equation $t \approx r$, or the logical negation $\neg(t \approx r)$ of a ground equation. Then $bt^+\varphi = eq_s(t,r) \approx 1$ or $bt^+\varphi = -eq_s(t,r) \approx 1$ for some sort s different from *newbool*. Thus, if $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$, we have

$$\Gamma \vdash t \approx r \quad \text{iff} \quad \mathbf{In}\,\mathsf{K} \models eq_s(t,r) \approx 1, \quad \text{and} \quad \Gamma \vdash \neg(t \approx r) \quad \text{iff} \quad \mathbf{In}\,\mathsf{K} \models eq_s(t,r) \approx 0.$$

(For ground identities, $\Gamma \models_{\mathrm{Min}} t \approx r$ if and only if $\Gamma \vdash t \approx r$.) This gives a way of characterizing, in terms of the structure of the Boolean part of the initial algebra of $\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$, when Γ is an initial, final, or complete specification of a particular data type.

Recall the definitions of Ω_Σ and ω_Σ given just after Lemma 2.3. ω_Σ is a finite conjunction of ground Σ-equations that is satisfied by a Σ-algebra $\mathbf{A}$ if and only if $\mathbf{Min}\,\mathbf{A}$ if trivial. Hence Γ has no trivial model if and only if $\Gamma \models_{\mathrm{Min}} \neg\omega_\Sigma$.

By a *co-atom* of a Boolean algebra $\mathbf{B}$ we mean an element b such that $b < 1$ and there exists no a such that $b < a < 1$. b is a co-atom if and only if, for every $a \in B$, either $a \leq b$ or $a + b = 1$.

COROLLARY 4.11. *Let Γ be any set of universal Σ-sentences, and let $\mathbf{A}$ be a Σ-data type. Let* $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGET}^+)$.

(i) *Γ is a weak initial specification of $\mathbf{A}$ if and only if, for every ground Σ-equation $t \approx r$,*

$$\mathbf{A} \models t \approx r \quad \textit{iff} \quad \mathbf{In}\,\mathsf{K} \models eq_s(t,r) \approx 1.$$

(ii) Γ *is an initial specification of* $\mathbf{A}$ *if and only if* $\mathbf{A} \models \Gamma$ *and, for every ground* Σ*-equation* $t \approx r$,

$$\mathbf{A} \models t \approx r \quad \textit{implies} \quad \mathbf{In}\,\mathsf{K} \models eq_s(t,r) \approx 1.$$

(iii) Γ *is a final specification of* $\mathbf{A}$ *if and only if* $\mathbf{A} \models \Gamma$ *and, for every ground* Σ*-equation* $t \approx r$,

$$\mathbf{A} \not\models t \approx r \quad \textit{implies} \quad \mathbf{In}\,\mathsf{K} \models eq_s(t,r) \leq \omega_\Sigma^*.$$

(iv) Γ *is a complete specification of* $\mathbf{A}$ *if and only if* $\mathbf{A}$ *is nontrivial,* $\mathbf{A} \models \Gamma$, *and* $(\omega_\Sigma^*)^{\mathbf{In}\,\mathsf{K}}$ *is a co-atom in* $(\mathbf{In}\,\mathsf{K})_{newbool}$.

Proof. Parts (i) and (ii) are immediate consequences of Theorem 2.11(i), (ii).

(iii) By Theorem 2.11(iii), we have that Γ is a final specification of $\mathbf{A}$ if and only if $\mathbf{A}$ is a model of Γ and, for each ground Σ-equation, $\mathbf{A} \not\models t \approx r$ implies $\Gamma \cup \{t \approx r\} \models \omega_\Sigma$, which is equivalent to $\Gamma \vdash t \approx r \to \omega_\Sigma$. But $\Gamma \vdash t \approx r \to \omega_\Sigma$ if and only if $\mathbf{In}\,\mathsf{K} \models (t \approx r \to \omega_\Sigma)^* \approx 1$ if and only if $\mathbf{In}\,\mathsf{K} \models eq_s(t,r) \leq \omega_\Sigma^*$.

(iv) Let $\mathbf{B} = \mathbf{In}\,\mathsf{K}$, and let $b = (\omega_\Sigma^*)^{\mathbf{B}}$. Assume that Γ is a complete specification of $\mathbf{A}$. Then $\mathbf{A}$ is a nontrivial model of Γ. To show that b is a co-atom of $\mathbf{B}_{newbool}$ it suffices to prove that, for every $a \in B_{newbool} \setminus \{0,1\}$, either $a \leq b$ or $a + b = 1$. By Lemma 3.4 we have

$$a = e_0 \cdots e_{n-1} + f_0 \cdots f_{m-1} + \cdots + g_0 \cdots g_{p-1},$$

where each of the primitive factors $e_i, f_j, \cdots, g_k$ is of the form $eq_s(t,r)^{\mathbf{B}}$ or $-eq_s(t,r)^{\mathbf{B}}$. Let h be any one of these primitive factors. By (ii) and (iii) we have that either $h \in \{0,1\}$, or $h \leq b$, or $-h \leq b$, i.e., $h + b = 1$. Without loss of generality we can assume that none of the primitive factors is 0 or 1. If each of the summands of a contains a primitive factor h such that $h \leq b$, then $a \leq b$. Otherwise, for all the primitive factors h of at least one of the summands, we have $h + b = 1$; thus $a + b = 1$, and so, b must be a co-atom.

Assume now, conversely, that $\mathbf{A}$ is a nontrivial model of Γ and b is a co-atom of $\mathbf{B}_{newbool}$. Let $t \approx r$ be any ground s-equation. Since $t \approx r$ holds in every trivial algebra, $b \leq eq(t,r)^{\mathbf{B}}$. So either $eq_s(t,r)^{\mathbf{B}} = b$ or $eq_s(t,r)^{\mathbf{B}} = 1$. Suppose $\mathbf{A} \models t \approx r$. Then $eq_s(t,r)^{\mathbf{A}^+} = 1$. Since $\mathbf{A}$ is nontrivial, $(\omega_\Sigma^*)^{\mathbf{A}^+} = 0$. Hence $eq_s(t,s)^{\mathbf{B}} \neq b$, and so $eq_s(t,r)^{\mathbf{B}} = 1$. This shows that Γ is an initial specification of $\mathbf{A}$. Now suppose $\mathbf{A} \not\models t \approx r$, i.e., $eq_s(t,r)^{\mathbf{A}^+} = 0$. Then $eq_s(t,r)^{\mathbf{B}} \neq 1$; hence $eq_s(t,r)^{\mathbf{B}} = b$, and so, a priori, $eq_s(t,r)^{\mathbf{B}} \leq b$. Thus Γ is a final specification of $\mathbf{A}$. □

COROLLARY 4.12. *Let* Γ *and* K *be as in Corollary* 4.11 *and assume that* $\mathbf{A}$ *is a model of* Γ. *If* Γ *has no trivial models, then* Γ *is a universal complete specification of* $\mathbf{A}$ *with hidden sort and operations if and only if* $(\mathbf{In}\,\mathsf{K})_{newbool} = \mathbf{B}_2$.

Proof. If Γ has no trivial models, then $\Gamma \models \neg\omega_\Sigma$, and hence $\mathbf{B} \models bt^+\omega_\Sigma \approx 0$. From Corollary 4.11(iv) we get that Γ is a complete specification of $\mathbf{A}$ if and only if $\mathbf{A} \models \Gamma$ and zero is a co-atom of $(\mathbf{In}\,\mathsf{K})_{newbool}$, i.e., $(\mathbf{In}\,\mathsf{K})_{newbool} = \mathbf{B}_2$. □

Example 3.10 (continued). The fact that Δ is not a complete specification of **NATBAG** is reflected in the fact that $\mathbf{NATBAG}^+$, the initial algebra of $\mathsf{Mod}(bt^+ \Delta \cup \mathrm{AXGET}^+)$, has more than two elements in its Boolean part. In particular, the value of $eq_{set}(s_{00}, s_0)$ in $\mathbf{NATBAG}^+$ differs from both zero and one.

5. If-then-else algebras. We now augment the operations of equality-test algebras by adjoining the *if-then-else operation* $[_,_,_]_s$ corresponding to each non-Boolean

sort s. $[_,_,_]_s$ is an operation of arity $bool\, s\, s$ and target sort s. $[b,a,c]_s$ equals a if $b=1$, and c if $b=0$.

The simplest notion of an if-then-else algebra is a homogeneous algebra with a 4-ary operation $[_,_,_,_]$ where $[a,b,c,d]^{\mathbf{A}}$ equals c if $a=b$, and d otherwise. These algebras (or, more precisely, the algebras polynomially equivalent to them) have been extensively studied in the literature of universal algebra. The varieties generated by classes of if-then-else algebras in this sense are called *discriminator varieties* (see Werner [46] and Burris and Werner [9]). Various other notions of an if-then-else algebra, both homogeneous and heterogeneous, have been considered in the literature of data types (see Bloom and Tindell [7], Guessarian and Meseguer [18], and Mekler and Nelson [33]). The particular notion of *if-then-else* (or ITE) *algebra* we consider here is a natural enrichment of an equality-test algebra.

With a few exceptions the theory of ITE algebras closely parallels that of equality-test algebras, the main difference being that the quasi variety and the variety generated by the class of ITE algebras coincide. The members of this variety are called *generalized if-then-else* (GITE) *algebras.* Applying the corresponding results of §3 we easily obtain a simple, finite, equational axiomatization of the variety of GITE algebras, and prove another analogue of the Stone representation theorem. Axiomatizations of the varieties generated by the various other versions of if-then-else algebras can be found in [7], [18], [33], [32], and [40].

All the results of §3 on the specification of ET algebras also have their analogues for ITE algebras, but with a significant improvement owing to the fact that the GITE algebras form a variety rather than just a quasi variety. For example, one of our main results is a simple algorithm for converting any finite universal initial specification of an ITE data structure into a finite-equational initial specification. We also discuss the equational specification of arbitrary data types using hidden sorts and operations. However, here the results are not so satisfactory. Given any finite universal initial specification of an arbitrary data structure $\mathbf{A}$, we can always transform it into a finite-equational initial specification of a GITE data structure $\mathbf{B}$ with the property that $\mathbf{A}$ is a subreduct of $\mathbf{B}$. But $\mathbf{A}$ is not, in general, the entire reduct of $\mathbf{B}$, so we do not normally get an equational specification of $\mathbf{A}$ with hidden sort and operations in the usual sense.

DEFINITION 5.1. (i) A signature Σ is called an **if-then-else signature** if it is an equality-test signature and, in addition, there exists an operation symbol $[_,_,_]_s :$ $bool\, s\, s \to s$ for each sort $s \neq bool$.

(ii) A Σ-algebra $\mathbf{A}$ is an **if-then-else (ITE) algebra** if Σ is an if-then-else signature, $\mathbf{A}$ is an equality-test algebra, and, for each $s \in S \setminus \{bool\}$ and all $b \in \mathbf{A}_{bool}$ and $a_0, a_1 \in \mathbf{A}_s$,

$$[b,a_0,a_1]_s^{\mathbf{A}} = \begin{cases} a_0, & \text{if } b = 1^{\mathbf{A}}; \\ a_1, & \text{otherwise (i.e., } b = 0^{\mathbf{A}}). \end{cases}$$

The class of all ITE algebras is denoted by ITE_Σ, or simply ITE.

Every if-then-else signature is a priori an equality-test signature, and every ITE algebra is a priori an equality-test algebra. ITE algebras are called *consistent equality-compatible algebras* in Padawitz [40].

$[_,_,_]_s$ is called the *s-if-then-else operation.* As in the case of equality-tests, no fundamental if-then-else operation for the Boolean sort is postulated since it can be defined in terms of the Boolean operations by

$$[x,y,z]_{bool} = x\cdot y + -x\cdot z.$$

A simple modification of the GET axioms gives a finite set of equational axioms for the variety generated by the ITE algebras.

DEFINITION 5.2. Let Σ be an if-then-else signature, and let AXGITE_Σ be the set of equations obtained from the axioms AXGET_Σ for GET Σ-algebras by replacing the one conditional axiom,

(Axget_6) $eq_s(x,y) \approx 1 \rightarrow x \approx y$,

by the following two equational axioms for each $s \in S \setminus \{bool\}$:

(Axgite_{6a}) $[1,x,y]_s \approx x$;

(Axgite_{6b}) $[eq_s(x,y),x,y]_s \approx y$.

Any Σ-algebra satisfying AXGITE_Σ is called a **generalized if-then-else** (**GITE**) **algebra**. The variety of GITE Σ-algebras is denoted by GITE_Σ.

Let $\mathbf{A}$ be an ITE algebra, s any sort (possibly Boolean), and $a,b \in A_s$. If $a = b$, then $eq_s^{\mathbf{A}}(a,b) = 1^{\mathbf{A}}$, and hence $[eq_s^{\mathbf{A}}(a,b),a,b]_s^{\mathbf{A}} = a = b$; otherwise $eq_s^{\mathbf{A}}(a,b) = 0^{\mathbf{A}}$ and $[eq_0^{\mathbf{A}}(a,b),a,b]_s^{\mathbf{A}}$ again equals b. So (Axgite_{6b}) is valid in $\mathbf{A}$, and, clearly, so is (Axgite_{6a}). Thus every ITE algebra is also a GITE algebra. Conversely, let $\mathbf{A}$ be any GITE algebra such that $\mathbf{A}_{bool} = \mathbf{B}_2$. Then ($\mathrm{Axgite}_{6a}$) and ($\mathrm{Axgite}_{6b}$) together imply $[_,_,_]_s^{\mathbf{A}}$ is the if-then-else operation. Hence $\mathbf{A}$ is an ITE algebra.

THEOREM 5.3. *Let Σ be any if-then-else signature.*

(i) $\mathsf{GITE}_\Sigma \subseteq \mathsf{GET}_\Sigma$.

(ii) $\mathsf{GITE}_\Sigma \cap \mathsf{ET}_\Sigma = \mathsf{ITE}_\Sigma$.

Proof. (i) We must verify that the conditional equation $eq_s(x,y) \approx 1 \rightarrow x \approx y$ holds in every GITE algebra. Let $\mathbf{A} \in \mathsf{GITE}$, and consider any $a,b \in A_s$ such that $eq_s^{\mathbf{A}}(a,b) = 1^{\mathbf{A}}$. Then using ($\mathrm{Axgite}_{6a}$) and ($\mathrm{Axgite}_{6b}$) we get $a = [1^{\mathbf{A}},a,b]_s^{\mathbf{A}} = [eq_s^{\mathbf{A}}(a,b),a,b]_s^{\mathbf{A}} = b$.

(ii) Obviously, $\mathsf{ITE} \subseteq \mathsf{GITE} \cup \mathsf{ET}$. Conversely, if $\mathbf{A} \in \mathsf{ET}$, then $\mathbf{A}_{bool}$ has exactly two elements. So, if $\mathbf{A}$ is a GITE algebra, it must be an ITE algebra. □

Thus ITE can be axiomatized by adjoining the universal sentence (Axet) to AXGITE.

Since GITE is a variety, every homomorphic image of a GITE algebra is again a GITE and hence a GET algebra. Thus every congruence on a GITE algebra is a GET congruence. Consequently, we immediately get from Theorems 3.6 and 5.3(ii) the following theorem.

THEOREM 5.4. *A* GITE *algebra is simple* (*in the absolute sense*) *if and only if it is an* ITE *algebra.* □

This gives the following result, which is in striking contrast to the situation for ET data types; see the remarks following Theorem 3.14, in particular Example 3.15.

COROLLARY 5.5. *Let* $\mathbf{A}$ *be an* ITE *data type. Every initial specification, or more generally every weak initial specification, of* $\mathbf{A}$ *is complete.*

Proof. For the proof, apply Corollary 2.7. □

Since GITE is a variety and is included in GET, every subvariety of GITE is a relative subvariety of GET, and hence all the results of §3 pertaining to relative subvarieties of GET can be automatically applied to these varieties. In particular, we get the following analogue of the Stone representation theorem.

THEOREM 5.6. (i) $\mathsf{K} = \mathbf{IP}_{\mathrm{SD}}(\mathsf{K} \cap \mathsf{ITE})$ *for any subvariety* K *of* GITE.

(ii) $\mathsf{GITE} = \mathbf{IP}_{\mathrm{SD}}(\mathsf{ITE})$.

Proof. Since K is a subvariety of GITE, it is a priori a relative subvariety of GET. Thus, by Theorem 3.7(i), each $\mathbf{A} \in \mathsf{K}$ is subdirectly representable in $\mathsf{K} \cap \mathsf{ET}$, which coincides with $\mathsf{K} \cap \mathsf{ITE}$ by Theorem 5.3(ii). This gives (i), and (ii) is a special case of (i). □

COROLLARY 5.7. $\mathsf{Qv}(\mathsf{ITE}) = \mathsf{Va}(\mathsf{ITE}) = \mathsf{GITE} = \mathsf{Mod}(\mathrm{AXGITE})$. □

Theorem 3.9 continues to hold when K is taken to be any subvariety of GITE and the class ET is replaced everywhere in the statement of the theorem by ITE.

A result very close to Corollary 5.7 can be found in Padawitz [40, Thm. 2.4].

5.1. Transforming the universal specifications into equational complete specifications. The notion of the Boolean transform of a universal sentence is well defined for any if-then-else signature since it is a priori an equality-test signature. And since every ITE algebra $\mathbf{A}$ is a priori an ET algebra, it follows trivially from Lemma 3.11 that, for any universal sentence φ, $\mathbf{A} \models \varphi$ if and only if $\mathbf{A} \models bt\,\varphi$. Combining this result with 5.6, we get the following analogues of Theorems 3.13 and 3.14 by essentially the same arguments.

THEOREM 5.8. *Let Γ be an arbitrary set of universal sentences.*

$$\mathsf{Va}\big(\mathsf{Mod}(\Gamma \cup \mathrm{AXGITE} \cup \{(\mathrm{Axet})\})\big) = \mathsf{Mod}(bt\,\Gamma \cup \mathrm{AXGITE}). \qquad \square$$

THEOREM 5.9. *Let Γ be any set of universal sentences, and let $\mathbf{A}$ be an* ITE *data type. If Γ is either an initial, weak initial, or a final specification of $\mathbf{A}$, then $bt\,\Gamma \cup \mathrm{AXGITE}$ is an equational complete specification of $\mathbf{A}$.* □

5.2. Equational specifications with hidden sorts and operations. The enrichments of an arbitrary signature Σ and Σ-algebra $\mathbf{A}$ to an if-then-else signature Σ^+ and ITE algebra $\mathbf{A}^+$ are defined in the obvious way. In the sequel, the superscript $^+$ will always mean a notion defined relative to the if-then-else enrichment. Although the results of §4 can be only partially extended in this context, we do get the following complete analogue of Theorem 4.3, and by essentially the same proof.

THEOREM 5.10. *For any signature Σ and any set Γ of universal sentences,*

$$\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGITE}^+) \cap \mathsf{ITE}^+ = \{\, \mathbf{A}^+ : \mathbf{A} \in \mathsf{Mod}\,\Gamma \,\}. \qquad \square$$

Similarly, we get complete analogues of Theorems 4.4 and 4.5(i). In fact, both results continue to hold as stated when AXGET^+ and ET^+ are replaced, respectively, by AXGITE^+ and ITE^+.

On the other hand, the second part of Theorem 4.5, together with Theorem 4.6, does not carry over completely. The proofs of these results depend on the fact that none of the hidden operations of the equality-test enrichment of Σ has a visible sort as target. This is obviously not a property of the if-then-else enrichment of Σ, and consequently if-then-else enrichments are not always sufficiently complete in the sense of [19] and [20]. The best we can do, in general, is the following.

THEOREM 5.11. *Let Σ be any signature, and let Γ be an arbitrary set of universal Σ-sentences:*

$$\mathsf{Qv}(\mathsf{Mod}\,\Gamma) = \mathbf{S}\big(\mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGITE}^+)|_\Sigma\big). \qquad \square$$

THEOREM 5.12. *Let Γ be any set of universal Σ-sentences, and $\mathbf{A}$ a Σ-data type. If Γ is an initial specification, or more generally a weak initial specification, of $\mathbf{A}$, then $\mathbf{A} \cong \mathsf{Min}((\mathbf{In}\,\mathsf{K})|_\Sigma)$, where $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \mathrm{AXGITE}^+)$.* □

Example 3.10 (continued). Let $\mathsf{K} = \mathsf{Mod}(bt^+\,\Delta \cup \mathrm{AXGITE}^+)$. $\mathbf{NATBAG} \neq \mathbf{In}\,\mathsf{K}|_\Sigma$ because

$$[eq_{set}(s_{00}, s_0), s_0, s_1]^{\mathbf{In}\,\mathsf{K}}_{set}$$

is an element of $\mathbf{NATBAG}^+_{bool}$ that is not contained in $\mathbf{NATBAG}_{bool}$.

Suppose in Theorem 5.12 that $\mathbf{In}\,\mathsf{K} \in \mathsf{ITE}^+$, i.e, $(\mathbf{In}\,\mathsf{K})_{newbool} \cong \mathbf{B}_2$. Then we will always get $[b, a, c]_s^{\mathbf{In}\,\mathsf{K}} \in \{a, c\}$ since b can only be zero or one. Thus $\mathsf{Min}((\mathbf{In}\,\mathsf{K})|_\Sigma) = (\mathbf{In}\,\mathsf{K})|_\Sigma$, and so $\mathbf{A} \cong (\mathbf{In}\,\mathsf{K})|_\Sigma$. Hence we do have that $bt^+\Gamma \cup \text{AXGITE}^+$ is an initial specification of $\mathbf{A}$ with hidden sort and operations in this special case. This allows us to obtain the following complete analogue of Theorem 4.7.

THEOREM 5.13. *Let* $\mathbf{A}$ *be a* Σ*-data type, and let* Γ *be a universal initial specification of* $\mathbf{A}$ *such that* Γ *has no trivial models. Let* $\mathsf{K} = \mathsf{Mod}(bt^+\Gamma \cup \text{AXGITE}^+)$. *The following are equivalent.*

(i) $bt^+\Gamma \cup \text{AXGITE}^+$ *is an equational complete specification of* $\mathbf{A}$ *with hidden sort and operations;*

(ii) $bt^+\Gamma \cup \text{AXGITE}^+$ *is an equational initial specification of* $\mathbf{A}^+$*;*

(iii) Γ *is a complete specification of* $\mathbf{A}$. □

Example 2.13 (continued). The equations of $bt^+\,\Delta$ given in §4, together with AXGITE^+, give an equational complete specification of **NATSTK** with $\mathbf{B}_2$ as a hidden sort and the equality-test and if-then-else operations as hidden operations.

Finally, we mention that all the results on the encoding of the universal theories in equational theories obtained at the end of §4 automatically apply to GITE algebras. In particular, Theorem 4.10 and Corollary 4.11 continue to hold with AXGET^+ replaced everywhere by AXGITE^+. This encoding, together with some of its important consequences, was first discovered by McKenzie [32] in the context of discriminator varieties.

6. Conclusions. We have seen that, with regard to equality-test algebras, the method of initial specification is no more powerful than complete (i.e., simultaneous initial and final) specification. This can be viewed as evidence of the inherent limitations of initial semantics, or that only the computable data domains of a data type can be expected to have an equality test. (A particular data domain, of sort s, say, is *computable* if the ground s-identities are recursive.) The results of §4 suggest that one way around this problem may be to consider generalized equality-test algebras with complex Boolean data domains, but with the property that $eq_s^{\mathbf{A}^+}$ is an equality test, i.e., it takes only the value zero or one, for every computable domain of sort s. Theorem 4.6 can be easily extended to show that any universal initial specification of $\mathbf{A}$ that is complete (in the obvious sense) on computable data domains can be converted into a conditional initial specification of a GET enrichment of $\mathbf{A}$ with this property. This leads to an interesting question concerning the results of Bergstra and Tucker [3] mentioned in §2 (in the remarks following Corollary 2.12). Recall that they have have shown that every semicomputable data type has a finite-equational initial specification with a hidden sort and operations, and every computable data type has a finite-equational complete specification with hidden operations. Can these two results be combined to show that every semicomputable data type has a finite initial specification (with hidden sort and operations) that is complete on data domains that are computable?

In [39] we extend the investigations of this paper to equality-test algebras in which the equality-tests may take values in a multiple-valued logic different from classical two-valued logic.

Acknowledgments. The author thanks the members of the Department of Mathematics, Statistics, and Computer Science of the University of Illinois at Chicago for their help and encouragement while the first draft of this paper was being written. Thanks are also due to several anonymous referees for helpful remarks and suggestions.

REFERENCES

[1] J.A. Bergstra and J.V. Tucker, *The completeness of the algebraic specification methods for computable data types*, Inform. and Control, 54(1982), pp. 186–200.

[2] ———, *Initial and final algebra semantics for data type specification: Two characterization theorems*, SIAM J. Comput., 12(1983), pp. 366–387.

[3] ———, *Algebraic specifications of computable and semicomputable data types*, Theoret. Comput. Sci., 50(1987), pp. 137–181.

[4] J.A. Bergstra, M. Broy, J.V. Tucker, and M. Wirsing, *On the power of algebraic specifications*, in Mathematical Foundations of Computer Science (Strbske Pleso, Czechkoslovakia, 1981), J. Gruska and M. Chytil, eds., Lecture Notes in Computer Science 118, Springer-Verlag, Berlin, 1981, pp. 193–204.

[5] A. Bertoni, G. Mauri, and P.A. Miglioli, *A characterization of abstract data types as model-theoretic invariants*, in Automata, Languages and Programming, 6th Colloquium (Graz, Austria), H.A. Maurer, ed., Lecture Notes in Computer Science, 71 Springer-Verlag, Berlin, 1979, pp. 26–38.

[6] G. Birkhoff, *On the structure of abstract algebras*, Proc. Cambridge Philos. Soc., 31(1935), pp. 433–454.

[7] S.L Bloom and R. Tindell, *Varieties of "if-then-else"*, SIAM J. Comput., 12(1983), pp. 677–707.

[8] M. Broy, W. Dosch, H. Parsch, P. Pepper, and M. Wirsing, *Existential quantifiers in abstract data types*, in Automata, Languages and Programming, 6th Colloquium (Graz, Austria), H.A. Maurer, ed., Lecture Notes in Computer Science 71, Springer-Verlag, Berlin, 1979, pp. 72–87.

[9] S. Burris and H. Werner, *Sheaf constructions and their elementary properties*, Trans. Amer. Math. Soc., 248(1979), pp. 269–309.

[10] R.M. Burstall, *Proving properties of programs by structural induction*, Comput. J., 12(1969), pp. 41–48.

[11] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification* 1 : *Equations and Initial Semantics*, Springer-Verlag, Berlin, 1985.

[12] V. Giarrantana, F. Gimona, and U. Montanari, *Observability concepts in abstract data type specifications*, in Mathematical Foundations of Computer Science, 5th Conference, Lecture Notes in Computer Science 45, Springer-Verlag, 1976, pp. 576–586.

[13] J.A. Goguen, *How to prove algebraic inductive hypotheses without induction: With applications to the correctness of data type representations*, in Automated Reasoning, 5th Conference, W. Bibel and R. Kowalski, eds., Lecture Notes in Computer Science 87, Springer-Verlag, Berlin, 1980, pp. 356–373.

[14] J.A. Goguen and J. Meseguer, *Completeness of many-sorted equational logic*, Houston J. Math., 11(1985), pp. 307–334.

[15] J.A. Goguen, J.W. Thatcher, and E. Wagner, *An initial algebra approach to the specification, correctness, and implementation of abstract data types*, in Current Trends in Programming Methodology, R. Yeh, ed., Prenctice-Hall, Englewood Cliffs, NJ, 1978, pp. 80–149.

[16] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, *Abstract data types as initial algebras and the correctness of data representations*, in Computer Graphics, Pattern Recognition and Data Structure, Conference Proceedings, Association for Computing Machinery, New York, 1975, pp. 89–93.

[17] G. Grätzer, *Universal Algebra,* Second Edition, Springer-Verlag, New York, 1979.

[18] I. Guessarian and J. Meseguer, *On the axiomatization of "if-then-else"*, SIAM J. Comput., 16 (1987), pp. 332–357.

[19] J.V. Guttag, *The specification and application to programming of abstract data types*, Ph.D. thesis, Report CSRG-59, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1975.

[20] J.V. Guttag and J.J. Horning, *The algebraic specification of abstract data types*, Acta Inform., 10 (1978), pp. 27–52.

[21] J. Hsiang and N. Dershowitz, *Rewrite methods for clausal and non-clausal theorem proving*, in Proc. International Conference on Automata, Languages and Programming, Barcelona, Spain, 1983, Lecture Notes in Computer Science 154, pp. 331–346.

[22] G. Hornung and P. Raulefs, *Terminal algebra semantics and retractions for abstract data types*, in Automata, Languages and Programming, 7th Colloquium (Noordwijkerhout, 1980), J.W. de Bakker and J. van Leeuwen, eds., Lecture Notes in Computer Science 85, Springer-Verlag, Berlin, 1980, pp. 310–325.

[23] S.N. KAMIN, *Final data type specifications: A new data type specification method*, in Principles of Programming Languages, 7th Conference (Las Vegas, NV, 1980), Association for Computing Machinery, 1980, pp. 131–138.

[24] D. KAPUR AND M.K. SRIVAS, *Expressiveness of the operation set of a data specification*, in Principles of Programming Languages, 7th Conference (Las Vegas, NV, 1980), Association for Computing Machinery, 1980, pp. 139–153.

[25] B. LISKOV AND S. ZILLES, *Specification techniques for data abstractions*, IEEE Trans. Software Engrg., 1(1975), pp. 7–19.

[26] J. ŁOŚ, *Quelques remarques théorèms et problèmes sur les classes définissables d'algèbras*, in Mathematical Interpretation of Formal Systems, North-Holland, Amsterdam, 1955, pp. 98–113.

[27] B. MAHR AND J.A. MAKOWSKY, *Characterizing specifications that admit initial semantics*, in Colloque sur les Arbres en Algebre et en Programmation (CAAP), 8th Colloquium, Lecture Notes in Computer Science 159, Springer-Verlag, New York, 1983, pp. 300–316.

[28] J.A. MAKOWSKY, *Why Horn formulas matter in computer science. Initial structures and generic examples*, J. Comput. System Sci., 34(1987), pp. 266–292.

[29] A.I. MAL'CEV, *The Metamathematics of Algebraic Systems. Collected Papers.* 1936–1967, North-Holland, Amsterdam, 1971.

[30] ———, *Algebraic Systems*, Springer-Verlag, Heidelberg, 1973.

[31] G. MARONGIU AND S. TULIPANI, *Finite algebraic specifications of semicomputable data types*, in TAPSOFT '87, Vol. 1 (Pisa, 1987), Lecture Notes in Computer Science 250, Springer-Verlag, Berlin, 1987, pp. 111–122.

[32] R. MCKENZIE, *On spectra, and the negative solution of the decision problem for identities having a finite nontrivial model*, J. Symbol. Logic, 40(1975), pp. 186–196.

[33] A. MEKLER AND E. NELSON, *Equational bases for if-then-else*, SIAM J. Comput., 16(1987), pp. 465–485.

[34] J. MESEGUER AND J.A. GOGUEN, *Initiality, induction, and computability*, in Algebraic Methods in Semantics, M. Nivat and J. Reynolds, eds., Cambridge University Press, Cambridge, U.K., 1985, pp. 459–540.

[35] L.S. MOSS, J. MESEGUER, AND J.A. GOGUEN, *Final algebras, cosemicomputable algebras, and degrees of unsolvability*, in Category Theory and Computer Science (Edinburgh, Scotland, 1987), Lecture Notes in Computer Science 283, Springer-Verlag, Berlin, New York, 1987, pp. 158–181.

[36] E. PAUL, *A new interpretation of the resolution rule*, in Proc. Seventh International Conference on Automated Deduction, (Napa, CA, 1984), R.E. Shostak, ed., Lecture Notes in Computer Science 170, Springer-Verlag, New York.

[37] ———, *On solving the equality problem in theories defined by Horn clauses*, in Proc. Eurocal '85, European Conference on Computer Science (Linz, Austria, April 1–3 1985), Vol. 2, B.F. Caviness, ed., Lecture Notes in Computer Science 204, Springer-Verlag, Berlin, 1985, pp. 363–377.

[38] D. PIGOZZI, *Finite basis theorems for relatively congruence-distributive quasivarieties*, Trans. Amer. Math. Soc., 310(1988), pp. 499–533.

[39] ———, *Data types over multiple-valued logics*, Theoret. Comput. Sci., 77(1990), pp. 161–194.

[40] P. PADAWITZ, *The equational theory of parameterized specifications*, Inform. and Comput., 76(1988), pp. 121–137.

[41] M.H. STONE, *Boolean algebras and their application to topology*, Proc. Nat. Acad. Sci. U.S.A., 20(1934), pp. 197–202.

[42] A. TARSKI, *Equational logic and equational theories of algebras*, in Contributions to Mathematical Logic (Colloquium, Hannover, Germany, 1966), H.A. Schmidt, K. Schütte, and H.-J. Thiele, eds., North-Holland, Amsterdam, 1968, pp. 275–288.

[43] J.W. THATCHER, E.G. WAGNER, AND J.B. WRIGHT, *Specification of abstract data types using conditional axioms*, IBM Research Report RC-6214, IBM Research Division, Yorktown Heights, NY, September, 1976.

[44] ———, *Data type specification: Parameterization and the power of specification techniques*, ACM Trans. Program. Lang. Syst., 4(1982), pp. 711–732.

[45] M. WAND, *Final algebra semantics and data type extension*, J. Comput. Systems Sci., 19(1979), pp. 27–44.

[46] H. WERNER, *Discriminator varieties*, Studien zur Algebra und ihre Anwendugen 6, Akademie Verlag, Berlin, 1978.

[47] M. WIRSING AND M. BROY, *Abstract data types as lattices of finitely generated models*, in Mathematical Foundations of Computer Science, 9th Symposium (Rydzyna, Poland, 1980), M. Dembiński, ed., Lecture Notes in Computer Science 88, Springer-Verlag, Berlin, 1980,

pp. 673–685.

[48] M. Broy, M. Wirsing, and C. Pair, *A systematic study of models of abstract data types*, Theoret. Comput. Sci., 33(1984), pp. 139–174.

[49] S. Zilles, *Algebraic specification of data types*, Project MAC Progress Report 11, Massachusetts Institute of Technology, Cambridge, MA, 1974.

[50] ———, *An introduction to data algebras*, working paper, IBM Research Laboratory, San Jose, CA, 1975.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 807-823, October 1991
© 1991 Society for Industrial and Applied Mathematics
001

THE MAXIMUM SIZE OF DYNAMIC DATA STRUCTURES*

CLAIRE M. KENYON-MATHIEU† AND JEFFREY SCOTT VITTER‡

Abstract. This paper develops two probabilistic methods that allow the analysis of the maximum data structure size encountered during a sequence of insertions and deletions in data structures such as priority queues, dictionaries, linear lists, and symbol tables, and in sweepline structures for geometry and Very-Large-Scale-Integration (VLSI) applications. The notion of the "maximum" is basic to issues of resource preallocation. The methods here are applied to combinatorial models of file histories and probabilistic models, as well as to a non-Markovian process (algorithm) for processing sweepline information in an efficient way, called "hashing with lazy deletion" (HwLD). Expressions are derived for the expected maximum data structure size that are asymptotically exact, that is, correct up to lower-order terms; in several cases of interest the expected value of the maximum size is asymptotically equal to the maximum expected size. This solves several open problems, including longstanding questions in queueing theory. Both of these approaches are robust and rely upon novel applications of techniques from the analysis of algorithms. At a high level, the first method isolates the primary contribution to the maximum and bounds the lesser effects. In the second technique the continuous-time probabilistic model is related to its discrete analog—the maximum slot occupancy in hashing.

Key words. analysis of algorithms, hashing, lazy deletion, maximum, queueing theory, Markov process, occupancy distribution, data structures, file histories, priority queues, dictionaries, lists, symbol tables, sweepline, computational geometry, VLSI

1. Introduction. The size attained by data structures is fundamental to issues of resource allocation, yet, until recently little was known about analyzing the *maximum* size attained over a period of time, which is important for preallocating resources. A possible explanation of this deficiency is that classical methods of analysis with generating functions and recurrences cannot be applied readily for the maximum function. In this paper we develop two asymptotic methods to study the distribution of the maximum size of data structures. The methods are robust in that they apply to several different combinatorial and probabilistic models. We also study a non-Markovian process called *hashing with lazy deletion* (HwLD), which corresponds to an efficient way of processing sweepline information in computational geometry and Very-Large-Scale-Integration (VLSI) layout applications [14].

One of the motivations for our study is the need to develop and analyze practical space-efficient plane-sweep algorithms. Some work in this area has been done by Van Wyk and Vitter [14]; Morrison, Shepp, and Van Wyk [11]; Mathieu and Vitter [9]; and Ottmann and Wood [12], but as the latter point out: "Surprisingly there has been little theoretical investigation of space-economical plane-sweep algorithms even though such algorithms have significant practical applications." Ottmann and Wood [12] do not investigate the maximum data structure size (that is, the maximum number of items cut by the sweepline); they express the running times of their algorithms in terms of the maximum size. Our approach in this paper is to examine the distribution of the maximum data structure size, based upon several popular input models, and in addition to show that the HwLD algorithm introduced is optimum simultaneously for both average running time and preallocated space.

---

* Received by the editors December 26, 1989; accepted for publication October 14, 1990. An extended abstract of this work appears in [10].

† Laboratoire d'Informatique de l'Ecole Normale Supérieure, 45, rue d'Ulm, 75230 Paris Cedex 05, France.

‡ Department of Computer Science, Brown University, Box 1910, Providence, Rhode Island 02912.
Support was provided in part by a National Science Foundation grant and by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM.

Data structures process a sequence of items over time; at time t the data structure stores the items that are "living" at time t. Let us think of the ith item as being an interval $[s_i, t_i]$ in the unit interval, containing a unique key k_i of supplementary information. The ith item is "born" at time s_i, "dies" at time t_i, and is "living" when $t \in [s_i, t_i]$. The data structure also handles dynamic queries over time. Let us denote the data structure size at time t by $Size(t)$. If we think of the items as horizontal intervals, then $Size(t)$ is just the number of intervals "cut" by the vertical line at position t. In a typical planesweep application, having to do with VLSI artwork analysis, we might have 10^6 intervals in the time range $[0, 1]$, with $E(Size) = 10^3$; that is, only square roots of the total number of items tend to be present at any given time [13]. Thus, for space efficiency, it is important to use a dynamic data structure whose size follows the growth rate of $Size(t)$.

In HwLD, items are stored in a hash table of H buckets, based upon the hash value of the key. The distinguishing feature of HwLD is that an item is not deleted as soon as it dies; the "lazy deletion" strategy deletes a dead item only when a later insertion accesses the same bucket. The number H of buckets is chosen so that the expected number of items per bucket is small. HwLD is thus more time-efficient than doing "vigilant-deletion," at a cost of storing some dead items.

To model insertions, deletions, and queries, we consider two classes of models: *combinatorial* models and *probabilistic* models. The combinatorial models are the discrete-time models of file histories introduced in [4], [5] to model the evolution of several classical types of dynamic data structures, such as priority queues, dictionaries, stacks, and linear lists. The second class of models consists of probabilisitic continuous-time models in equilibrium, in which the birthtimes of items are independent and form a Poisson process with birth rate λ. Various models of lifetime are considered. Not only does our approach work for these models, but it can also be adapted to handle models in which lifetimes are not independent, such as the $M/M/1$ probabilistic model and the non-Markovian models corresponding to HwLD.

We denote by $Use(t)$ the number of items stored at time t in the HwLD data structure. The lazy deletion strategy means that $Use(t) \geqq Size(t)$. Let t^* be any time t that maximizes $E(Size(t))$. (For the probabilistic models, $E(Size(t))$ is the same for all t.) Van Wyk and Vitter [14] compute $E(Size(t^*))$ and show, via generating function and approximation techniques, that $E(Use(t^*)) \sim E(Size(t^*)) + H$ for the combinatorial model of priority queues and for the $M/M/\infty$ probabilistic model. Big-oh bounds on $E(\max_{t\in[0,1]}\{Size(t)\})$ and $E(\max_{t\in[0,1]}\{Use(t)\})$ were only recently obtained by Mathieu and Vitter [9] under certain assumptions for the $M/G/\infty$ probabilistic model. Exact formulas were also developed for several combinatorial and probabilistic models that could be used to compute the distribution of $\max_{t\in[0,1]}\{Size(t)\}$ numerically, but they do not seem to give any asymptotic information. However, the fact that the relevant transform in each case was expressed simply as the ratio of consecutive classical orthogonal polynomials gave informal evidence that some common asymptotic method(s) might exist to analyze the different models.

In this paper we develop general asymptotic methods using techniques from analysis of algorithms to settle the open problems posed in [14], [11], and [9]. We derive *asymptotically exact* expressions for $E(\max_{t\in[0,1]}\{Size(t)\})$ and $E(\max_{t\in[0,1]}\{Use(t)\})$ for several combinatorial and probabilistic models. In particular we show that HwLD is *asymptotically optimal in terms of preallocated storage.* The gist of our first method is to concentrate on the primary contribution to the maximum and to show via probabilistic techniques that the rest of the contribution is negligible. The hard part is coping with the inherent lack of independence of the size as a function

of time. We show exactly when the expected maximum size is asymptotically more than the maximum expected size and when they are equal. The second method we use, for the continuous-time probabilistic models, is a discrete counterpart having to do with the maximum slot occupancy in hashing. This approach offers another illustration of the strong connections between discrete and continuous models in the analysis of algorithms.

2. Analysis of combinatorial models. File histories, as introduced in [4], [5] model the evolution of several classical types of dynamic data structures, including priority queues (PQ), dictionaries (D), symbol tables (ST), stacks (S), and linear lists (LL). The data structures are treated as combinatorial objects; their performance characteristics are determined by the *relative* order of the elements they contain, not by the actual values of the elements. Thus, we say that there are $k+1$ ways of inserting a new element into a dictionary of size k, since there are $k+1$ "gaps" where the new element can fit in, relative to the k elements already present. The evolution of the data structure is represented as a path in $\mathbf{Z}^2$, where the x-coordinate counts the number of operations, whether they be insertions, deletions, or queries, and the y-coordinate counts the size. Each step is of the type $(a, b) \to (a+1, b \pm 1)$ (insertion or deletion) or $(a, b) \to (a+1, b)$ (positive or negative query). To each step we associate a certain choice among the possibilities. For example, in priority queues, deletions can be performed only for the minimum element, so the number of possibilities for a deletion is one. The probability model is that all possibilities are equally likely, with the constraint that the data structure is empty initially and at the end. The following table summarizes the number of possibilities for each type of data structure and operation, in terms of the current data structure size k:

	PQ	D	LL	ST	S
Insertions	$k+1$	$k+1$	$k+1$	$k+1$	1
Deletions	1	k	k	1	1
Positive queries	0	k	0	k	0
Negative queries	0	$k+1$	0	0	0

As an introduction to our first method, let us consider the combinatorial model of file histories corresponding to the size of priority queues (PQ). An equivalent formulation, as considered in [14], is to generate the $2n$ birthtimes and deathtimes of the n items as independent uniform random variates in the unit interval $[0, 1]$. The ith item is born at time $\min\{s_i, t_i\}$ and dies at time $\max\{s_i, t_i\}$. The average priority queue size $E(Size(t)) = 2nt(1-t)$ varies parabolically in the unit interval and attains its maximum $n/2$ when $t = \frac{1}{2}$, as shown in Fig. 1.

This "peak" in the value of $E(Size(t))$ suggests that the value $\max_{t\in[0,1]}\{Size(t)\}$ should be achieved in a neighborhood of $t = \frac{1}{2}$ and thus should be $\sim n/2$. This was conjectured in [14]. In this section we introduce our first method and use it to prove the conjecture. The method will be developed further in the next section for the probabilistic models, where the expected values in question are flat and have no peaks like the ones considered in this section.

THEOREM 2.1. *For priority queue and dictionary file histories of length* $2n$, *we have*

$$E(\max\{Size(t)\}) \sim \max_{t\in[0,1]}\{E(Size(t))\} = \frac{n}{2}.$$

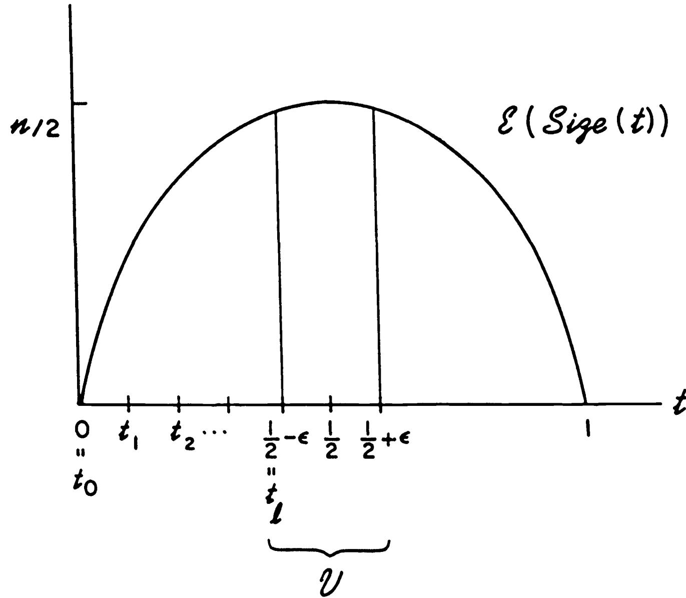


FIG. 1. *Graph of $E(Size(t))$, as a function of t, for the combinatorial model of priority queues considered in* § 2. *The quantity $Size(t)$ has a geometric interpretation as the number of items (horizontal intervals) "cut" by the vertical line (sweepline) at position t. (The graph is bell-shaped, and this makes the analysis of $E(\max_{t\in[0,1]}\{Size(t)\})$ easier; however, this is not the case for the "flat" distributions of the probabilistic models in* § 3.)

THEOREM 2.2. *For priority queue and dictionary file histories of length* $2n$, *if* $H = o(n)$ *we have*

$$E\left(\max_{t\in[0,1]}\{Use(t)\}\right) \sim \frac{n}{2}.$$

Proof of Theorem 2.1. First we prove Theorem 2.1 for priority queues. Since we have $E(\max_{t\in[0,1]}\{Size(t)\}) \geqq E(Size(\frac{1}{2})) = n/2$, our main problem is to show the other direction, namely, that $E(\max_{t\in[0,1]}\{Size(t)\}) \lesssim E(Size(\frac{1}{2}))$.[1]

We consider the neighborhood $\mathcal{V} = (\frac{1}{2}-\varepsilon, \frac{1}{2}+\varepsilon)$ of $t = \frac{1}{2}$, where $\varepsilon = n^{-1/6}$, as pictured in Fig. 1. We shall prove that with high probability the maximum M of $Size(t)$ is reached for $t \in \mathcal{V}$. Let v denote the number of births and deaths in interval $\mathcal{V}$. If the maximum is reached inside $\mathcal{V}$, then its value is at most $Size(\frac{1}{2}) + v$; thus we have

$$E(M) \leqq E(Size(\tfrac{1}{2})) + E(v) + n \cdot \Pr\{M \text{ reached outside } \mathcal{V}\}. \tag{2.1}$$

The first two terms are clearly equal to $n/2 + 4n\varepsilon \sim n/2$. All that remains is to prove that $\Pr\{M \text{ reached outside } \mathcal{V}\} = o(1)$. By symmetry, this probability can be bounded by

$$2 \cdot \Pr\{\exists t \in [0, \tfrac{1}{2}-\varepsilon], Size(t) > Size(\tfrac{1}{2})\}. \tag{2.2}$$

The problem is that the values of $Size(t)$ at two different t are clearly not independent. In order to get around this problem, we divide $[0, \frac{1}{2}-\varepsilon]$ into $l = n(\frac{1}{2}-\varepsilon)$ equal-sized

[1] We adopt the notation $f(n) \lesssim g(n)$ (as $n\to\infty$) if there is a function $h(n)$ such that $h(n) \sim g(n)$ (as $n\to\infty$) and $f(n) \leqq h(n)$ for all $n \geqq 1$.

intervals $[t_i, t_{i+1}]$, with endpoints $t_i = i/n$, for $0 \leqq i \leqq l$. Let $a = n/2 - n^{26/50}$ and $a' = n/2 - 2n^{26/50}$. We have

$$\begin{aligned}(2.3)\qquad &\Pr\left\{\exists t \leqq \frac{1}{2} - \varepsilon,\ Size(t) > Size\left(\frac{1}{2}\right)\right\} \\ &\leqq \Pr\left\{Size\left(\frac{1}{2}\right) < a\right\} + \Pr\left\{\exists t \leqq \frac{1}{2} - \varepsilon,\ Size(t) > a\right\} \\ &\leqq \Pr\left\{Size\left(\frac{1}{2}\right) < a\right\} + \sum_{0 \leqq i \leqq l} \Pr\{Size(t_i) > a'\} \\ &\quad + \sum_{0 \leqq i \leqq l} \Pr\{x_i > a - a'\},\end{aligned}$$

where x_i is the number of births in interval $[t_i, t_{i+1}]$. Since $\Pr\{Size(t_i) > a'\}$ is maximized at $t_i = \frac{1}{2} - \varepsilon$, and since $x_1, x_2, \cdots$ are identically distributed, we get

$$\begin{aligned}(2.4)\qquad &\Pr\left\{\exists t \leqq \frac{1}{2} - \varepsilon,\ Size(t) > Size\left(\frac{1}{2}\right)\right\} \\ &\leqq \Pr\left\{Size\left(\frac{1}{2}\right) < a\right\} + \frac{n}{2} \cdot \Pr\left\{Size\left(\frac{1}{2} - \varepsilon\right) > a'\right\} \\ &\quad + \frac{n}{2} \cdot \Pr\{x_0 > a - a'\}.\end{aligned}$$

We analyze each term separately. For fixed t, the distribution of $Size(t)$ is well known. It is binomially distributed with parameter $t(1-t)$:

$$\Pr\{Size(t) = k\} = \binom{n}{k} (2t(1-t))^k (1 - 2t(1-t))^{n-k};$$

$$E(Size(t)) = 2nt(1-t);$$

$$\mathrm{Var}\,(Size(t)) = 2nt(1-t)(1-2t(1-t)).$$

Thus Chebyshev's inequality yields

$$\Pr\left\{Size\left(\frac{1}{2}\right) < \frac{n}{2} - 2n^{26/50}\right\} < \frac{1}{4} n^{-2/50} = o(1).$$

As for the next term, we have

$$\Pr\left\{Size\left(\frac{1}{2} - \varepsilon\right) > \frac{n}{2} - 2n^{26/50}\right\} = \sum_{k > n/2 - 2n^{26/50}} \binom{n}{k} \left(\frac{1}{2} - 2\varepsilon^2\right)^k \left(\frac{1}{2} + 2\varepsilon^2\right)^{n-k}.$$

The terms of the sum form a decreasing sequence, and the ratio between two successive terms is at most

$$\left(\frac{n/2 + 2n^{26/50}}{n/2 - 2n^{26/50}}\right)\left(\frac{\frac{1}{2} - 2\varepsilon^2}{\frac{1}{2} + 2\varepsilon^2}\right) = \left(\frac{\frac{1}{2} + 2n^{-24/50}}{\frac{1}{2} - 2n^{-24/50}}\right)\left(\frac{\frac{1}{2} - 2n^{-1/3}}{\frac{1}{2} + 2n^{-1/3}}\right) < 1.$$

The sum can thus be replaced by a geometric sum, and after computing its asymptotics using Stirling's formula, we find that

$$\frac{n}{2} \cdot \Pr\left\{Size\left(\frac{1}{2} - \varepsilon\right) > a'\right\} \leqq n^{5/6} e^{-8n^{1/3}} = o(1).$$

The third term is also computed easily, since

$$\Pr\{x_0 = k\} = \binom{2n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{2n-k}.$$

Using Chebyshev's inequality, we find again that

$$\frac{n}{2}\Pr\{x_0 > n^{26/50}\} = o(1).$$

Thus the theorem is proved for priority queues.

The proof for dictionaries goes along the same lines. Let $H_{k,l,n}$ be the number of histories going from level k to level l in n steps. If we consider the size after m steps of a dictionary history going from level 0 to 0 in n steps, its distribution is given by

$$\Pr\{Size(m) = i\} = \frac{H_{0,i,m}H_{i,0,n-m}}{H_{0,0,n}}.$$

We know from [4] that

$$\sum_{k,l,n} H_{k,l,n}u^k v^l \frac{z^n}{n!} = \frac{1}{1 - z(1+u)(1+v) - uv}.$$

We deduce that

$$\Pr\{Size(m) = i\} = \frac{\binom{m}{i}\binom{n-m}{i}}{\binom{n}{m}}.$$

The expected value is

$$E(Size(m)) = n\frac{m}{n}\left(1-\frac{m}{n}\right).$$

Thus the graph of $E(Size(m))$, when m goes from 0 to n, describes a parabola, just as for priority queues. Since we know the distribution of $Size(m)$, we can from then on work out a proof exactly similar to the priority queue case (further details are omitted). □

Proof of Theorem 2.2. For Theorem 2.2, as in Theorem 2.1, the lower bound $n/2$ applies. To get an upper bound, we use the bound

$$(2.5)\quad E\left(\max_{t\in[0,1]}\{Use(t)\}\right) \leqq H\cdot E\left(\max_{t\in[0,1]}\{Use_1(t)\}\right) = H\cdot E\left(\max_{t\in[0,1]}\{Size_1(t)\}\right),$$

where $Size_1(t)$ is the number of living items present in the first bucket of the HwLD table at time t. The inequality in (2.5) follows from the uniformity of the hash function in HwLD, and the subsequent equality follows from the fact that the priority queue is initially empty and thus $Use_1(t)$ and $Size_1(t)$ attain the same maxima. Let n_1 be the number of living items that hash to bucket 1 in HwLD. We have

$$(2.6)\quad \Pr\{n_1 = k\} = \binom{n}{k}\left(\frac{1}{H}\right)^k\left(1-\frac{1}{H}\right)^{n-k}.$$

We can now use the result of the previous theorem:

$$(2.7)\quad \begin{aligned} E\left(\max_{t\in[0,1]}\left\{Size_1(t)\right\}\right) &\lesssim \sum_{k>n/2H}\left(\frac{k}{2}\cdot\Pr\{n_1 = k\}\right) \\ &\quad + \frac{n}{2H}\cdot\Pr\left\{n_1 \leqq \frac{n}{2H}\right\}, \end{aligned}$$

from which we find, after some elementary calculations, that

$$E\left(\max_{t\in[0,1]}\{Size_1(t)\}\right) \lesssim \frac{n}{2H}. \tag{2.8}$$

By summing on the H buckets, we get $E(\max_{t\in[0,1]}\{Use(t)\}) \lesssim n/2$. The proof for dictionaries is similar. □

These techniques can also be applied to the combinatorial models for other types of file histories, namely, linear lists and symbol tables, to get similar results. (Note that the techniques cannot be used for stacks, because the random variable "maximum size" is not concentrated enough around its expected value in that case.)

3. Analysis of the probabilistic models. In this section we consider continuous-time probabilistic models in equilibrium. The birthtimes of items are independent and form a Poisson process with intensity λ; the probability of a birth during a time interval of length Δt is $\sim \lambda \Delta t$, as $\Delta t \to 0$. In the first model we consider, the lifetimes of the items are independently distributed according to an arbitrary distribution with mean $1/\mu$. An important special case is when the lifetimes satisfy the memoryless exponential distribution. This model and its special case are well known and are referred to as the stationary $M/G/\infty$ and $M/M/\infty$ models.

The maximum is more difficult to analyze for the probabilistic models, since the expected value of $Size(t)$ is no longer "peaked" around a certain value of t. An easy analysis (see Feller [1], for example) shows that for each t the distribution of $Size(t)$ is Poisson with mean λ/μ. But the maximum value of $Size(t)$ in each case is sufficiently concentrated about its mean so that our method is applicable.

THEOREM 3.1. *In the stationary $M/M/\infty$ model with birth rate λ and average lifetime $1/\mu$, we have, assuming either that $\mu \to 0$ or that $\mu = \Omega(1)$ and $\lambda \to \infty$,*

$$E\left(\max_{t\in[0,1]}\{Size(t)\}\right) \sim \begin{cases} \dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu)\to 0; \\[2ex] d\dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu)\to c; \\[2ex] \dfrac{f(\lambda,\mu)}{\ln f(\lambda,\mu)}\,\dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu)\to\infty, \end{cases}$$

where $f(\lambda,\mu) = (\ln \lambda)/(\lambda/\mu)$ and the constant $d \geqq 1$ is defined implicitly from the constant c by $d \ln d - d = c - 1$. In the more general $M/G/\infty$ case, in which the lifetime distribution can be arbitrary, the asymptotic upper bounds hold; in the first case $\ln \lambda = o(\lambda/\mu)$, the corresponding lower bound is trivial, so we get asymptotic equality.

THEOREM 3.2. *In the stationary $M/G/\infty$ model, if $\ln \lambda = o(\lambda/\mu)$, we have*

$$E\left(\max_{t\in[0,1]}\{Use(t)\}\right) \sim \frac{\lambda}{\mu} + H.$$

The condition $\ln \lambda = o(\lambda/\mu)$ is typically met in practice in geometry applications, as in [13]. Similar results for $E(\max_{t\in[0,1]}\{Use(t)\})$ hold as for Cases 2 and 3 of Theorem 3.1, except that the conditions are more intricate.

Proof of Theorem 3.1. If $\mu \to 0$, then $\lambda = o(\lambda/\mu)$, and

$$Size(0) \leqq \max_{t\in[0,1]}\{Size(t)\} \leqq Size(0) + \#[0,1],$$

where $\#[0, 1]$ denotes the number of births in $[0, 1]$. Taking expectations we get

$$\frac{\lambda}{\mu} \leqq E\left(\max_{t\in[0,1]} \{Size(t)\}\right) \leqq \frac{\lambda}{\mu} + \lambda,$$

and hence $E(\max_{t\in[0,1]} \{Size(t)\}) \sim \lambda/\mu$.

From now on, we assume that $\lambda \to \infty$ and that $\mu \geqq \alpha$ for some positive constant α. First we derive the upper bounds for $E(\max_{t\in[0,1]} \{Size(t)\})$. We use the basic identity

$$E\left(\max_{t\in[0,1]} \{Size(t)\}\right) = \sum_{k\geqq 1} \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} \geqq k\right\}. \tag{3.1}$$

The probabilites in the sum form a decreasing sequence. We are going to show that $\max_{t\in[0,1]} \{Size(t)\}$ has a distribution concentrated near some value V (to be specified later). We trivially have

$$E\left(\max_{t\in[0,1]} \{Size(t)\}\right) \leqq V(1+2\varepsilon) + \sum_{k>V(1+2\varepsilon)} \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} \geqq k\right\}. \tag{3.2}$$

The desired upper bound $E(\max_{t\in[0,1]} \{Size(t)\}) \lesssim V$ follows if we show, for an adequate choice of $\varepsilon \to 0$, that $\sum_{k>V(1+2\varepsilon)} \Pr\{\max_{t\in[0,1]} \{Size(t)\} \geqq k\} = o(1)$.

In order to evaluate the probabilities, we have to deal with the lack of independence of the successive values of $Size(t)$, as t goes from 0 to 1. We partition $[0, 1]$ into N intervals of equal size, $I_0 = [t_0, t_1)$, $I_1 = [t_1, t_2), \cdots, I_j = [t_j, t_{j+1}), \cdots$. The number of intervals N will be defined to be λ in Case 1 and $(\lambda/\varepsilon V)^{1+\varepsilon}$ in Cases 2 and 3. The key point is that $\max_{[0,1]} \{Size(t)\} > k$, where $k > V(1+2\varepsilon)$, only if there is an interval endpoint t_i where $Size(t_i) \geqq (k+V)/2$ or if one of the intervals has at least $(k-V)/2$ births:

$$\begin{aligned} \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} \geqq k\right\} &\leqq \Pr\left\{ \exists j,\ Size(t_j) \geqq \frac{k+V}{2}\right\} \\ &\quad + \Pr\left\{ \exists j,\ \# I_j \geqq \frac{k-V}{2}\right\}, \end{aligned} \tag{3.3}$$

where $\# I_j$ denotes the number of births during time interval I_j. By (3.2) we get

$$\begin{aligned} E\left(\max_{t\in[0,1]} \{Size(t)\}\right) &\leqq V(1+2\varepsilon) + 2N \sum_{k>V(1+\varepsilon)} \Pr\{Size(0) \geqq k\} \\ &\quad + 2 \sum_{k>\varepsilon V} \Pr\{\exists j,\ \# I_j \geqq k\}. \end{aligned} \tag{3.4}$$

For the $M/M/\infty$ process, the random variables $Size(0)$ and $\# I_j$ are Poisson distributed with means λ/μ and λ/N, respectively. The rest of the proof consists of technical computations and approximations, with adequate choices for the parameters V, N, and ε.

First we compute

$$S = 2 \sum_{k>\varepsilon V} \Pr\{\exists j,\ \# I_j \geqq k\} \leqq 2N \sum_{k>\varepsilon V} \Pr\{\# I_0 \geqq k\}. \tag{3.5}$$

The inequality holds because the random variables $\# I_j$ are identically distributed. The Poisson probability function of $\# I_0$ is

$$\Pr\{\# I_0 = k\} = e^{-\lambda/N} \frac{(\lambda/N)^k}{k!}. \tag{3.6}$$

The ratio between consecutive probabilities is less than $\lambda/(N\varepsilon V)=o(1)$, for suitable choices of N, ε, and V. By (3.6) we get

$$\frac{\Pr\{\#I_0=k+1\}}{\Pr\{\#I_0=k\}}=\frac{\lambda}{N(k+1)}<\frac{\lambda}{N\varepsilon V}=o(1),$$

for a suitable choice of N and V. Thus we have

$$\Pr\{\#I_0\geqq k\}=\sum_{j\geqq k}\Pr\{\#I_0=j\}\leqq\frac{1}{1-\lambda/(N\varepsilon V)}\cdot\Pr\{\#I_0=k\},$$

and by (3.5),

$$S\leqq\frac{2N}{1-\lambda/(N\varepsilon V)}\sum_{k>\varepsilon V}\Pr\{\#I_0=k\}\leqq\frac{2N}{(1-\lambda/(N\varepsilon V))^2}\cdot\Pr\{\#I_0=\varepsilon V\}.$$

Substituting (3.6), we get for large λ

$$S\leqq 3Ne^{-\lambda/N}\frac{(\lambda/N)^{\varepsilon V}}{(\varepsilon V)!}.$$

If we pick V so that $V\geqq(\lambda/\mu)\to\infty$, we can choose $\varepsilon\to 0$ so that $\varepsilon V\to\infty$. We now apply Stirling's approximation formula to get

$$S\leqq\frac{3N}{\sqrt{\varepsilon V}}e^{-\lambda/N}\left(\frac{\lambda e}{N\in V}\right)^{\varepsilon V}. \tag{3.7}$$

Our choices of N and V for the three cases of Theorem 3.1 are as follows:

Case 1. Assuming that $\ln\lambda=o(\lambda/\mu)$, we fix $N=\lambda$ and $V=(\lambda/\mu)$. We find, if $\varepsilon\to 0$ slowly enough, that $S=o(1)$.

Case 2. Assuming that $\ln\lambda=c(\lambda/\mu)$, where c is a positive constant, we fix

$$N=\left(\frac{\lambda}{\varepsilon V}\right)^{1+\varepsilon}\quad\text{and}\quad V=d\frac{\lambda}{\mu}(1+\sqrt{\varepsilon}),$$

where d is the solution of $d\ln d-d=c-1$. For $\varepsilon\to 0$ slowly enough we find that $S=o(1)$.

Case 3. Assuming that $f(\lambda,\mu)=(\ln\lambda)/(\lambda/\mu)\to\infty$, we fix

$$N=\left(\frac{\lambda}{\varepsilon V}\right)^{1+\varepsilon}\quad\text{and}\quad V=\frac{f}{\ln f}\frac{\lambda}{\mu}.$$

The analysis works as in Case 2 to show that $S=o(1)$.

We now turn our attention to bounding the other sum in (3.4), namely,

$$P=N\sum_{k>V(1+\varepsilon)}\Pr\{\mathit{Size}(0)\geqq k\}. \tag{3.8}$$

We want to show that $P=o(1)$. The random variable $\mathit{Size}(0)$ has the Poisson distribution:

$$\Pr\{\mathit{Size}(0)=k\}=e^{-\lambda/\mu}\frac{(\lambda/\mu)^k}{k!}. \tag{3.9}$$

The ratio between two successive terms is at most $(\lambda/\mu)/(V(1+\varepsilon))$ when $k>V(1+\varepsilon)$. Thus we can write

$$P\leqq N\Big/\left(1-\frac{\lambda}{\mu}\Big/(V(1+\varepsilon))\right)^2\cdot\Pr\{\mathit{Size}(0)=V(1+\varepsilon)\}. \tag{3.10}$$

By using (3.9), the fact that $V \geqq (\lambda/\mu)$ in all cases, and Stirling's formula, we get for large V,

$$P \leqq \frac{N}{\varepsilon^2} e^{-\lambda/\mu} \frac{(\lambda/\mu)^{V(1+\varepsilon)}}{(V(1+\varepsilon))!} \leqq \frac{N}{\varepsilon^2} \frac{e^{-\lambda/\mu}}{\sqrt{V}} \left(\frac{e\lambda/\mu}{V(1+\varepsilon)} \right)^{V(1+\varepsilon)}.$$

We can now evaluate this expression for all three cases of the theorem, and we find that, if ε goes to 0 slowly enough, we have in all cases $P = o(1)$, which concludes the proof of the upper bound of the theorem.

For the lower bound, let us for clarity restrict ourselves to Case 3 of the theorem, and assume that $\lambda = \mu$ (the proof in Case 2 is similar). The above proof yields $E(\max_{t\in[0,1]} \{Size(t)\}) \leqq ((\ln \lambda)/\ln \ln \lambda)(1+\varepsilon)$, for some positive $\varepsilon = o(1)$. We shall now show that the reverse also holds, namely, that $E(\max_{t\in[0,1]} \{Size(t)\}) \geqq ((\ln \lambda)/\ln \ln \lambda)(1-\varepsilon)$, for some positive $\varepsilon = o(1)$. We shall prove this by starting with (3.1) and showing that

$$\text{(3.11)} \qquad \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} \geqq k \right\} \sim 1, \quad \text{for } k \leqq \frac{\ln \lambda}{\ln \ln \lambda}(1-\varepsilon).$$

We once again partition [0, 1] into $l = \lambda^{1-\varepsilon}$ equal-sized intervals, with endpoints $t_i = i/\lambda^{1-\varepsilon}$, for $0 \leqq i \leqq l$. We have

$$\text{(3.12)} \qquad \begin{aligned} \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} < k \right\} &= \sum_{j_1,\cdots,j_l<k} \Pr\{\forall i, Size(t_i) = j_i\} \\ &= \sum_{j_1,\cdots,j_l<k} \prod_{1\leqq i\leqq l} \Pr\{Size(t_i) = j_i \mid Size(t_{i-1}) = j_{i-1}\}. \end{aligned}$$

The motivation for our choice of interval size is to have enough births and deaths in each interval so that the values of $Size(t)$ at the endpoints are "sufficiently independent." Let $P_n(t)$ denote $\Pr\{Size(t) = n \mid Size(0) = j_{i-1}\}$. We define the generating function $P(s, t) = \sum_n P_n(t) s^n$, which is equal to

$$\text{(3.13)} \qquad P(s, t) = e^{-(1-s)(1-e^{\mu t})}(1-(1-s)\, e^{-\mu t})^{j_{i-1}}$$

(cf. Feller [1]). The conditional probability term in (3.12) is $P_{j_i}(1/\lambda^{1-\varepsilon}) = \langle s^{j_i} \rangle P(s, 1/\lambda^{1-\varepsilon})$. By extracting the coefficient of s^{j_i} in (3.13) and using asymptotic approximations, we find that there is "sufficient independence":

$$\text{(3.14)} \qquad \Pr\left\{ \max_{t\in[0,1]} \{Size(t)\} < k \right\} \sim (\Pr\{Size(0) < k\})^{\lambda^{1-\varepsilon}} \sim o(1),$$

for $k \leqq ((\ln \lambda)/\ln \ln \lambda)(1-\varepsilon)$. By letting $\varepsilon \to 0$ at an appropriate rate, we prove our goal (3.11), which completes the proof. □

Proof of Theorem 3.2. The lower bound here is easy:

$$E\left(\max_{t\in[0,1]} \{Use(t)\} \right) \geqq E(Use(0)) = \frac{\lambda}{\mu} + H.$$

We prove the upper bound when $\ln \lambda = o(\lambda/\mu)$. First, we consider the $M/M/\infty$ case. From [13], the stationary distribution of $Use(t)$ is

$$\text{(3.15)} \qquad \Pr\{Use(t) = k\} = e^{-\lambda/\mu} \frac{(\lambda/\mu)^{k-H}}{(k-H)!}.$$

We partition the interval $[0, 1]$ into λ intervals of equal size, $I_j = [t_j, t_{j+1})$. Let $V = (\lambda/\mu) + H$. If $k > V(1+2\varepsilon)$, we have, with the same techniques used for the previous theorem,

$$\Pr\left\{\max_{t\in[0,1]}\{Use(t)\} \geqq k\right\} \leqq \lambda \cdot \Pr\left\{Use(0) \geqq \frac{k+V}{2}\right\} + \Pr\left\{\exists j, \# I_j \geqq \frac{k-V}{2}\right\}. \tag{3.16}$$

And the same approximations as above show that when $\ln \lambda = o(\lambda/\mu)$ we have

$$E\left(\max_{t\in[0,1]} Use(t)\right) \lesssim \frac{\lambda}{\mu} + H.$$

In order for the proof to work in the $M/G/\infty$ model, we need only show that the stationary distribution of $Use(t)$ is the same as in the $M/M/\infty$ model. We shall compute the probability

$$p_{m,n}(t) = \Pr\{Size(t) = m,\ Use(t) = m+n \mid Size(0) = Use(0) = 0\} \tag{3.17}$$

and let $t \to \infty$ to get the stationary distribution. We assume for simplicity that $H = 1$.

$$p_{m,n}(t) = \int_0^t \lambda\, e^{-\lambda t - x} \cdot \Pr\{Size(x) = m+n-1 \text{ and } n \text{ deaths in } (x, t)\}\, dx. \tag{3.18}$$

We assume that $x > \sqrt{t}$; the lower part of the integral is negligible. We have

$$\Pr\{Size(x) = m+n-1\} \sim e^{-\lambda/\mu} \frac{(\lambda/\mu)^{m+n-1}}{(m+n-1)!}. \tag{3.19}$$

Let $B(x)$ be the distribution of the service time. The probability that there are n deaths in (x, t) can be split into two terms, depending on whether the element born at time x dies before time t or not. The probability that a given element, alive at time x, dies before time t is equal to

$$p(x) = \frac{(1/x)\int_0^x (B(t-u) - B(x-u))\, du}{1 - (1/x)\int_0^x B(x-u)\, du}. \tag{3.20}$$

Thus we find that $p_{m,n}(t)$ is asymptotically equal to

$$\int_{\sqrt{t}}^t \lambda\, e^{-\lambda t - x}\, e^{-\lambda/\mu} \frac{(\lambda/\mu)^{m+n-1}}{(m+n-1)!} \cdot \left((1 - B(t-x))\binom{m+n-1}{n} p(x)^n (1-p(x))^{m-1} + B(t-x)\binom{m+n-1}{n-1} p(x)^{n-1}(1-p(x))^m\right) dx.$$

With standard asymptotics, we get

$$\Pr\{Use(t) = s\} = \sum_{m+n=s} p_{m,n}(t) \sim \int_{\sqrt{t}}^t \lambda\, e^{-\lambda t - x}\, e^{-\lambda/\mu} \frac{(\lambda/\mu)^{s-1}}{(s-1)!}\, dx \sim e^{-\lambda/\mu} \frac{(\lambda/\mu)^{s-1}}{(s-1)!}. \tag{3.21}$$

The case for general H is similar. Thus the stationary distribution of $Use(t)$ is the same for $M/G/\infty$ and $M/M/\infty$ processes, and Theorem 3.2 is proved. □

It is worthwhile noting that Theorem 3.1 derives results in queueing theory, using non-queueing theory techniques from the analysis of algorithms. By a simple change of scale, we can extend the range over which we take the maximum from the unit interval [0, 1] to [0, T]. The last subcase of Theorem 3.1 says that, if λ and μ are constant, then $E(\max_{t\in[0,T]}\{Size(t)\}) \sim \ln T/\ln\ln T$, which was a longstanding open problem. Our method also applies to the $M/M/1$ model: For constants λ and μ with $\lambda/\mu = c < 1$, we have $E(\max_{t\in[0,T]}\{Size(t)\}) \sim -\ln T/\ln c$, which previously had been proved only by Brownian motion techniques [2].

THEOREM 3.3. *In the stationary $M/M/1$ model, with birth rate λ and average lifetime $1/\mu$, we have*

$$E\left(\max_{t\in[0,1]}\{Size(t)\}\right) \sim \begin{cases} 1 & \text{if } \lambda = o(\mu^{2/3}); \\ \dfrac{\ln \lambda}{-\ln c} & \text{if } \lambda/\mu \to c < 1, \end{cases} \qquad \lambda \to \infty.$$

Proof of Theorem 3.3. Let us restrict ourselves to Case 2 (the more difficult case). The stationary distribution of the size of an $M/M/1$ process is well known (see [1]).

$$\Pr\{Size(t) = k\} = \frac{(\lambda/\mu)^k}{1-(\lambda/\mu)}.$$

For the upper bound, we divide [0 1] into λ intervals $I_i = [t_i t_{i+1})$, with $t_i = i/\lambda$, and use the same technique as before:

$$\Pr\left\{\max_{t\in[0,1]}\{Size(t)\} \geqq k\right\} \leqq \lambda \cdot \Pr\left\{Size(0) \geqq \frac{k+k_0}{2}\right\}$$
$$+\lambda \cdot \Pr\left\{\#I_0 \geqq \frac{k-k_0}{2}\right\}.$$

Fixing $k_0 = \ln \lambda/(-\ln c)$, we find after some calculations that

$$E\left(\max_{t\in[0,1]}\{\text{Size(t)}\}\right) \leqq \frac{\ln \lambda}{-\ln c}(1+o(1)).$$

The lower bound proof follows the proof in the $M/M/\infty$ case. □

4. Time hashing: The discrete analog. In this section we analyze $\max_{0\leqq t\leqq 1}\{Size(t)\}$ by relating the problem to its discrete version—the maximum slot occupancy in hashing. The tricky part is handling the lack of independence of slot occupancies.

THEOREM 4.1. *In the stationary $M/G/\infty$ model, with birth rate λ and average lifetime $1/\mu$, we have, assuming either that $\mu \to 0$ or that $\mu = \Omega(1)$ and $\lambda \to \infty$,*

$$E\left(\max_{t\in[0,1]}\{Size(t)\}\right) \lesssim \begin{cases} \dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu) \to 0; \\ d\dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu) \to c; \\ \dfrac{2f(\lambda,\mu)}{\ln f(\lambda,\mu)}\dfrac{\lambda}{\mu} & \text{if } f(\lambda,\mu) \to \infty, \end{cases}$$

where $f(\lambda,\mu) = (\ln \lambda)/(\lambda/\mu)$ and the constant $d \geqq 1$ is defined implicitly from the constant c by $d\ln d - d = 2c - 1$. When $f(\lambda,\mu) \to 0$, we have asymptotic equality.

It is interesting to note the close correspondence between the above formulas and the formulas for the maximum bucket occupancy in hashing, given in Kolchin, Sevast'yanov, and Chistyakov [8] (cf. § 4). The formulas for cases 2 and 3 of Theorem 4.1 are slightly weaker than the corresponding bounds in Theorem 3.1.

THEOREM 4.2. *In the stationary $M/G/\infty$ model, with birth rate λ and average lifetime $1/\mu$, we have, assuming that $\mu \to 0$ or $\lambda = o(H)$ or that $\mu = \Omega(1)$ and $\lambda \to \infty$,*

$$E\left(\max_{t\in[0,1]}\{Use(t)\}\right) \lesssim \begin{cases} \dfrac{\lambda}{\mu} + H & \text{if } f(\lambda,\mu)\to 0; \\ d\dfrac{\lambda}{\mu} + H & \text{if } f(\lambda,\mu)\to c; \\ \dfrac{2f(\lambda,\mu)}{\ln f(\lambda,\mu)}\dfrac{\lambda}{\mu} + H & \text{if } f(\lambda,\mu)\to\infty, \end{cases}$$

where $f(\lambda, \mu) = (\ln \lambda)/(\lambda/\mu)$ and the constant $d \geqq 1$ is defined implicitly from the constant c by $d \ln d - d = 2c - 1$. When $f(\lambda, \mu) \to 0$, we have asymptotic equality.

THEOREM 4.3. *In the stationary $M/G/\infty$ model, with birth rate λ and average lifetime $1/\mu$, assuming that $\ln \lambda = o(H)$, $\mu = \Omega(1)$, and $\lambda \to \infty$, we have*

$$E\left(\max_{t[0,1]}\{Use(t)\} - \max_{t\in[0,1]}\{Size(t)\}\right) \sim H.$$

Results similar to those in cases 2 and 3 of Theorems 4.1 and 4.2 also hold for $E(\max_{t\in[0,1]}\{Use(t)\} - \max_{t\in[0,1]}\{Size(t)\})$, except that the conditions are more complicated.

An approach called "time hashing" was introduced in [9] to give optimum bounds to within a constant factor for $E(\max_{t\in[0,1]}\{Size(t)\})$ when $f(\lambda, \mu) \to 0$ and $E(\max_{t\in[0,1]}\{Use(t)\} - \max_{t\in[0,1]}\{Size(t)\})$ when $H \geqq (\ln \lambda)^{1+\varepsilon}$, for constant $\varepsilon > 0$. The approach we use here to show that the constant factors are in fact 1 is the "inverse" of the approach used in [9], so a brief explanation of the former technique is called for.

For example, in the analysis of $E(\max_{0\leqq t\leqq 1}\{Size(t)\})$ when $\ln \lambda = o(\lambda/\mu)$ in [9], all the items that are alive for at least some time in $[0, 1]$ are considered. There are stages $k = 0, 1, 2, \cdots, K$, and each stage has an associated hash table. For $0 \leqq k \leqq K$, all items (intervals) born in $(-(1/\mu)2^k, 1]$ with lifespan in the range $((1/\mu)2^{k-1}, (1/\mu)2^k]$ are put into stage k; in addition, all such items with lifespan $\leqq (1/2\mu)$ are put into stage 0. Each stage consists of a hash table of $\mu 2^{-k} + 1$ slots. The jth slot, for $0 \leqq j \leqq \mu 2^{-k}$, represents the interval of time $((1/\mu)(j-1)2^k, (1/\mu)j2^k]$. An item in stage k is placed into the slot corresponding to its birthtime. A special stage $K+1$ is constructed to store all the items that do not fit into one of the earlier stages. The parameter K is chosen large enough so that the number of items in stage $K+1$ is $O(\lambda/\mu)$. The important link between this discrete version of the problem and the original continuous one is the following relation:

$$\max_{0\leqq t\leqq 1}\{Size(t)\} \leqq 2 \sum_{0\leqq k\leqq K+1} \max_{0\leqq j\leqq \mu 2^{-k}}\{N_k(j)\},$$

where $N_k(j)$ denotes the number of items in the jth slot of stage k. The bound $\max_{0\leqq t\leqq 1}\{Size(t)\} = O(\lambda/\mu)$ was proved by showing that $\max_{0\leqq j\leqq \mu 2^{-k}}\{N_k(j)\} = O(\lambda/(2^k\mu))$, under the assumption that $\ln \lambda = O(\lambda/\mu)$.

Proof of Theorem 4.1. We shall prove Theorem 4.1 simultaneously for all three cases, $f(\lambda, \mu) \to 0$, $\to c$, and $\to \infty$, by showing that $E(\max_{0\leqq t\leqq 1}\{Size(t)\}) \lesssim d\lambda/\mu$, where $d = d(\lambda, \mu)$ is the solution of the equation $d \ln d - d = 2f - 1$. In case 1, for example, we have $d \sim 1$, and in case 3 we have $d \sim 2f/\ln f$.

Note that for case 1 this upper bound will prove the asymptotic result claimed in Theorem 4.1, namely, $E(\max_{0\leqq t\leqq 1}\{Size(t)\})\sim(\lambda/\mu)$, since $E(\max_{0\leqq t\leqq 1}\{Size(t)\})\geqq Size(0)=\lambda/\mu$.

If $\mu\to 0$, then the result follows immediately, as noted in § 3. So we assume that $\lambda\to\infty$ and $\mu=\Omega(1)$. We "invert" the process used in [9] to prove the big-oh bound, as shown in Fig. 2. Instead of letting the stages of time hashing grow coarser and coarser, we consider the limiting case in which the slots represent smaller and smaller units of time relative to the item sizes. We use only one stage, but the number of slots varies with λ and μ. We use $m=g\mu$ slots, for any $g\to\infty$. (Without loss of generality we choose g so that $g\mu$ is an integer.) The jth slot, for $1\leqq j\leqq g\mu$, represents the time interval $((j-1)/(g\mu), j/(g\mu)]$. For each item we place an entry into each slot whose associated time interval intersects the item's lifetime. If we define $N(j)$ to be the slot occupancy of slot j, it is easy to see that the following upper bound holds.

LEMMA 4.1. *We have*

$$\max_{0\leqq t\leqq 1}\{Size(t)\}\leqq\max_{1\leqq j\leqq g\mu}\{N(j)\}.$$

To prove Theorem 4.1 it suffices to show that $E(\max_{1\leqq j\leqq g\mu}\{N(j)\})\lesssim d\lambda/\mu$.

LEMMA 4.2. *The slot occupancies $N(j)$ are Poisson-distributed with mean n/m, where $n=\lambda(g+1)$ is the average number of items inserted into the hash table of $m=f\mu$ slots.*

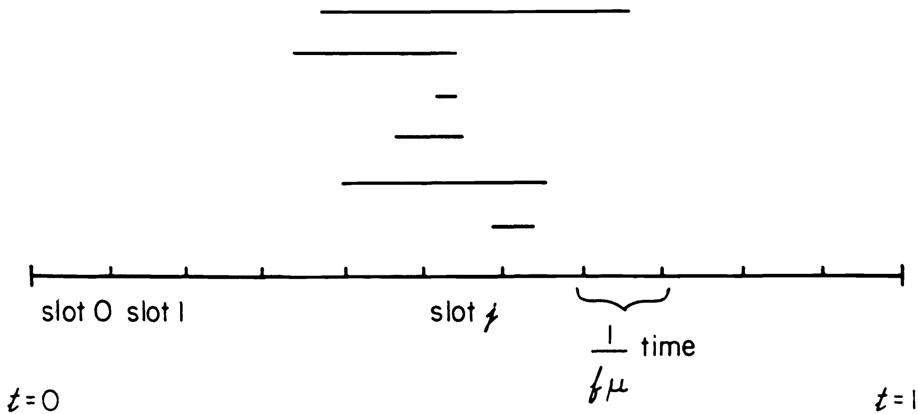


FIG. 2. *Typical items that contribute an entry to slot j in the time hashing table.*

Proof. The slot occupancy $N(j)$ of the jth slot is equal to the sum of two independent quantities: the number of items living at time $(j-1)/(g\mu)$ plus the number of items born during the time interval $((j-1)/(g\mu), j/(g\mu)]$. These two quantities are Poisson-distributed with means λ/μ and $\lambda/(g\mu)$, respectively. Hence, $N(j)$ is Poisson-distributed with mean $(\lambda/\mu)(1+(1/g))=n/m$. □

The following lemma is useful for studying the maximum slot occupancy in time hashing, because the random variables X_j are not required to be independent.

LEMMA 4.3 [9]. *For random variables $X_1,\cdots,X_m$, if we have $\Pr\{X_j>b\}\leqq 1/(nm)$, for all $1\leqq j\leqq m$, where $n=E(\sum_j X_j)$, then*

$$E\left(\max_{1\leqq j\leqq m}\{X_j\}\right)\leqq b+\frac{1}{n}\cdot E\left(\max_{1\leqq j\leqq m}\{X_j\}\,\middle|\,\max_{1\leqq j\leqq m}\{X_j\}>b\right).$$

Proof. We condition the expectation based upon if $\max_{1\leqq j\leqq m}\{X_j\}$ is $\leqq b$ or $>b$:

$$E\left(\max_{1\leqq j\leqq m}\{X_j\}\right)\leqq b\cdot \Pr\left\{\max_{1\leqq j\leqq m}\{X_j\}\leqq b\right\}$$

$$+E\left(\max_{1\leqq j\leqq m}\{X_j\}\,\middle|\,\max_{1\leqq j\leqq m}\{X_j\}>b\right)$$

$$\cdot \Pr\left\{\max_{1\leqq j\leqq m}\{X_j\}>b\right\}.$$

The first probability term is bounded trivially by 1, and the second is bounded by

$$\Pr\left\{\max_{1\leqq j\leqq m}\{X_j\}>b\right\}\leqq \Pr\{X_1>b\}+\Pr\{X_2>b\}+\cdots+\Pr\{X_m>b\}\leqq m\cdot\frac{1}{nm}=\frac{1}{n}.\quad\square$$

To apply Lemma 4.3 to prove Theorem 4.1, we use $X_j=N(j)$, $b=d(\lambda/\mu)(1+(1/g))$, $n=\lambda(g+1)$, and $m=g\mu$, where $g\to\infty$ slowly. (In particular, we require that $g\leqq(d-1)\lambda/\mu$.) We have

$$nm\cdot\Pr\{N(j)>b\}=nm\,e^{-n/m}\sum_{k>b}\frac{(n/m)^k}{k!}$$

$$\leqq\frac{n^2\,e^{-n/m}(n/m)^b}{b!(b+1-n/m)}$$

$$\leqq\frac{g^2}{\sqrt{2\pi(\lambda/\mu)d(1+1/g)}(d-1)(\lambda/\mu)(1+(1/g))}\cdot\frac{\lambda^2\,e^{(d-1)(\lambda/\mu)(1+1/g)}}{d^{d(\lambda/\mu)(1+1/g)}}\tag{4.1}$$

$$\leqq\frac{\lambda^2\,e^{(d-1)(\lambda/\mu)(1+1/g)}}{d^{d(\lambda/\mu)(1+1/g)}},$$

for large λ, by Stirling's formula. Taking logarithms of (4.1) and using the definition of d, we get

$$\ln(nm\cdot\Pr\{N(j)>b\})\leqq 2\ln\lambda-\frac{\lambda}{\mu}\left(1+\frac{1}{g}\right)(d\ln d-d+1)$$
$$=-\frac{2\ln\lambda}{g}<0.\tag{4.2}$$

This implies that the left-hand side of (4.1) is less than or equal to 1 for large λ, and hence the conditions for Lemma 4.3 are satisfied.

Lemma 4.3 gives us

$$\max_{1\leqq j\leqq f\mu}\{N(j)\}\leqq d\frac{\lambda}{\mu}\left(1+\frac{1}{g}\right)$$
$$+\frac{1}{n}E\left(\max_{1\leqq j\leqq g\mu}\{N(j)\}\,\middle|\,\max_{1\leqq j\leqq g\mu}\{N(j)\}>d\frac{\lambda}{\mu}\left(1+\frac{1}{g}\right)\right).\tag{4.3}$$

The random variables $N(j)$ are not independent, but the conditional expectation on the right-hand side of (4.3) can be bounded by

$$E\left(\max_{1\leqq j\leqq g\mu}\{N(j)\}\,|\,N(1)>d\frac{\lambda}{\mu}\left(1+\frac{1}{g}\right)\right)$$
$$\leqq n+E\left(N(1)\,|\,N(1)>d\frac{\lambda}{\mu}\left(1+\frac{1}{g}\right)\right)=O(n). \tag{4.4}$$

Plugging (4.4) back into (4.3) gives us $E(\max_{1\leqq j\leqq f\mu}\{N(j)\})\lesssim d(\lambda/\mu)$, which proves Theorem 4.1. □

Proof of Theorem 4.2. Theorem 4.2 can be proved in an identical way to Theorem 4.1. For each t, $Use(t)-H$ is Poisson-distributed with mean λ/μ [9]. The techniques in the proof can then be applied to $Use(t)-H$ instead of to $Size(t)$.

For example, let us define the time hashing as in the proof of Theorem 4.1, except that we account for lazy deletion. For each item, we place an entry into each slot whose associated time interval intersects the item's presence in the HwLD data structure. We define $N(j)$ to be the slot occupancy of slot j. We have the corresponding versions of Lemmas 4.1 and 4.2 in Lemma 4.4.

LEMMA 4.4. *We have*

$$\max_{0\leqq t\leqq 1}\{Use(t)\}\leqq \max_{1\leqq j\leqq g\mu}\{N(j)\}.$$

To prove Theorem 4.2 it suffices to show that $E(\max_{1\leqq j\leqq g\mu}\{N(j)\})\lesssim d\lambda/\mu+H$.

LEMMA 4.5. *The slot occupancies $N(j)-H$ are Poisson-distributed with mean n/m, where $n=\lambda(g+1)$ is the average number of items inserted into the hash table of $m=f\mu$ slots.*

The rest of the proof proceeds analogously as for Theorem 4.1. □

The proof of Theorem 4.3 is more complicated and is omitted for brevity. It uses the techniques developed in [9]. We bound $E(\max_{t\in[0,1]}\{Use(t)-Size(t)\})$ using Lemma 4.3 and an application of Chernoff's bound, which in turn gives us a bound for $E(\max_{t\in[0,1]}\{Use(t)\}-\max_{t\in[0,1]}\{Size(t)\})$, as desired.

5. Conclusions. We have developed two probabilistic methods that are useful for the analysis of the distribution of the maximum size of data structures. We get the asymptotic value of the expected maximum of $Size(t)$ and $Use(t)$ for several different combinatorial and probabilistic models of insertion and deletion. This solves the open problems from [14], [11], and [9] as well as a longstanding open problem from queueing theory.

In our first method we isolate the primary contribution to the maximum and bound the lesser effects. Our second technique, which we use for a continuous model, takes advantage of the close connections between the model and its discrete analog, namely, the maximum slot occupancy in hashing. These methods can be used to get estimates of second-order terms and higher moments of the expected maximum, as well as estimates of the shape of the distribution of the maximum. The techniques also appear directly applicable to the study of the maximum size of other dynamic data structures, such as quad trees, k-d trees, and radix-exchange tries.

REFERENCES

[1] W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. 1, Third Edition, John Wiley, New York, 1968.

[2] G. FAYOLLE, Personal communication, 1988.

[3] P. FLAJOLET, *Analyse d'algorithmes de manipulation d'arbres et de fichiers*, Cahiers Bureau Universitaire Rech. Opér., 34-35 (1981), pp. 1-209.

[4] P. FLAJOLET, J. FRANÇON, AND J. VUILLEMIN, *Computing integrated costs of operations with applications to dictionaries*, in Proc. 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, April/May 1979, pp. 49-61.

[5] ———, *Sequence of operations analysis for dynamic data structures*, J. Algorithms, 1 (1980), pp. 111-141; A shortened version appeared in Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, Puerto Rico, October 1979, pp. 183-195.

[6] S. KARLIN AND J. L. MCGREGOR, *The differential equations of birth-and-death processes, and the Stieltjes moment problem*, Trans. Amer. Math. Soc., 85 (1957), pp. 489-546.

[7] L. KLEINROCK, *Queueing Systems. Vol.* I: *Theory*, John Wiley & Sons, Inc., New York, 1975.

[8] V. F. KOLCHIN, B. A. SEVAST'YANOV, AND V. P. CHISTYAKOV, *Random Allocations*, V. H. Winston & Sons, Washington, DC, 1978.

[9] C. M. MATHIEU AND J. S. VITTER, *Maximum queue size and hashing with lazy deletion*, in Proc. 20th Annual Symposium on the Interface of Computing Science and Statistics, Reston, VA, April 1988.

[10] ———, *General methods for the analysis of the maximum size of dynamic data structures*, in Proc. 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy, July 1989.

[11] J. MORRISON, L. A. SHEPP, AND C. J. VAN WYK, *A queueing analysis of hashing with lazy deletion*, SIAM J. Comput., 16 (1987), pp. 1155-1164.

[12] T. OTTMANN AND D. WOOD, *Space-Economical Plane-Sweep Algorithms*, Computer Vision, Graphics, and Image Processing, 34 (1986), pp. 35-51.

[13] T. G. SZYMANSKI AND C. J. VAN WYK, *Space-Efficient Algorithms for* VLSI *Artwork Analysis*, in Proc. 20th IEEE Design Automation Conference, 1983, pp. 743-749.

[14] C. J. VAN WYK AND J. S. VITTER, *The complexity of hashing with lazy deletion*, Algorithmica, 1 (1986), pp. 17-29.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 824-833, October 1991
© 1991 Society for Industrial and Applied Mathematics
002

TIME COMPLEXITY OF BOOLEAN FUNCTIONS ON CREW PRAMS*

MIROSŁAW KUTYŁOWSKI†

Abstract. This paper is concerned with parallel random access machines (PRAMs), where each processor can read from and write into a common random access memory. Different processors may read the same memory location at a time, but only one processor is allowed to write into it (the CREW model). Suppose f is a Boolean function of n variables. Let $CREW(f)$ be the number of steps required by CREW PRAMs to compute function f. It has been proved that $CREW(f) \geqq \log_b crit(f)$, where $b \approx 4.79$ and $crit(f)$ is the critical complexity of f (see [S. Cook, C. Dwork, and R. Reischuk, *SIAM J. Comput.*, 15 (1986), pp. 87-97]). It was proved by Parberry and Pei Yuan Yan [*SIAM J. Comput.*, 20 (1991), pp. 88-99] that the same holds for $b = 4$. It follows that the time required by the logical "or" of n variables is at least $\log_4 n$. This paper presents an essentially different method of estimating PRAM complexity of Boolean functions. Let n_f be the number of inputs $x \in \{0, 1\}^n$ for which $f(x) = 1$. Let $i_f = \max\{j: 2^j \mid n_f\}$. Then $CREW(f) \geqq \log_c (n - i_f)$, where $c \approx 2.618$. Thanks to this result, the time complexity of the logical "or" of n variables is determined exactly. This in turn allows better estimations of time complexity of the threshold functions to be obtained. Another corollary is that for sorting n arbitrary keys, PRAMs require time, which can be determined up to five steps.

Key words. parallel computation, parallel random access machines, time bounds

AMS(MOS) subject classification. 68Q10

1. Introduction. Parallel random access machines (PRAMs) serve as a model of parallel computers communicating through a shared memory. They seem to be well suited to the task of the theoretical design of parallel algorithms. Different types of PRAMs have been already studied. We concern ourselves only with so-called CREW PRAMs (i.e., with Concurrent Read, Exclusive Write access to the common memory). We investigate the time required to compute Boolean functions on such devices.

Each PRAM consists of a collection of processors and common memory cells. Each computation consists of several computation steps. At each step, each processor can read from at most one memory cell; then it does some computing (arbitrarily long and deterministic); and finally, it has the possibility to write into a chosen memory cell. Any number of processors can read from a given memory cell simultaneously, but only one processor can write into a given memory cell during one computation step. This means that for the considered machines, write conflict does not occur. Usually, it requires a very careful design of the algorithm to get this condition fulfilled. We do not assume that the considered PRAMs are oblivious. In other words, for different input strings, different processors may write into a given memory cell at a particular moment of the computation (while for each input there is at most one such processor by the exclusive write assumption).

If PRAM M computes a function f of n arguments, then the arguments are stored initially in the first n memory cells (one argument in a cell). The result of a computation is given at the last moment of computation in some specially chosen memory cell.

* Received by the editors February 27, 1989; accepted for publication (in revised form) November 27, 1990. This research was supported by the Alexander von Humboldt Foundation of Bonn. This paper was written while the author was visiting the Institut für Theoretische Informatik, Technische Hochschule Darmstadt, Darmstadt, Germany.

† Institute of Computer Science, University of Wrocław, Przesmyckiego 20, 51-151, Wrocław, Poland. Present address, Universität-GH Paderborn FB 17-Mathematik, Informatik, Warburgerstrasse 100, Paderborn, Germany D-W-4790 (mirekk@uni-paderborn.de and mirekk@plwruw11.bitnet).

We place no restrictions on the number of processors and memory cells. In each memory cell we can store an arbitrary amount of data. There are potentially infinitely many internal states of each processor. More information about this and similar models of computation could be found, for instance, in [6], [2], [4]. These problems are also quite completely discussed in [10].

Suppose M is a CREW PRAM computing a Boolean function $f:\{0,1\}^n \to \{0,1\}$. By time complexity of M, $CREW(M)$, we understand the maximal number of steps required by M to compute $f(\mathbf{x})$ for $\mathbf{x} \in \{0,1\}^n$. Let $CREW(f) = \min\{CREW(M)\colon M \text{ computes } f\}$. We consider the following general problem.

PROBLEM 1.1. *Given a Boolean function f, determine* $CREW(f)$.

It might be said that the PRAMs that we consider are too "unrealistic." For instance, we should put a restriction on the number of processors and the maximal size of a word that may be stored by a single cell, if we wish to find any practical implementation of a PRAM. The reason why we consider PRAMs with such unrestricted resources is that we can prove the same lower bounds for these powerful machines. Unexpectedly, for many functions these lower time bounds are equal or very close to the upper bounds obtained for very restricted "realistic" PRAMs. For instance, it is true for the logical "or" (see below), PARITY, and many threshold functions ([7]).

A lower bound of $CREW(f)$ was determined in terms of $crit(f)$, the critical complexity of f. Recall that $crit(f)$ is the maximal number j such that there is an input $\mathbf{x} \in \{0,1\}^n$ and $J \subseteq \{1, 2, \cdots, n\}$, $|J| = j$ such that

$$\forall i \in J \qquad f(\mathbf{x}) \neq f(I(\mathbf{x}, i)),$$

where $I(\mathbf{x}, i)$ is the sequence, which differs from $\mathbf{x}$ only at position i (see [4], [3], [10]). For instance, if f is the logical "or" of n variables, then $crit(f) = n$. Indeed, if we change any bit of the input $0, 0, \cdots, 0$, then the value of the "or" changes. Recall the following result from [4].

THEOREM 1.2 [4]. *Suppose f is a Boolean function. Then* $CREW(f) \geqq \log_b crit(f)$, *where* $b = \frac{1}{2}(5+\sqrt{21}) \approx 4.79$. *In particular, if f is the logical "or" of n variables, then* $CREW(f) \geqq \log_b n$.

This result was improved in [8]. It was shown that the theorem also holds for $b = 4$. On the other hand, it was already shown in [4] that the theorem does not hold for $b < 3$.

Recall that the Fibonacci numbers are defined inductively as follows: $F_0 = 0$, $F_1 = 1$, $F_{m+2} = F_{m+1} + F_m$, for $m \geqq 0$. Let $\varphi(n) = \min\{j\colon F_{2t+1} \geqq n\}$ (then $\varphi(n) \approx \log_b n$ for $b = \frac{1}{2}(3+\sqrt{5})$). The following upper bound for the logical "or" was presented in [4].

THEOREM 1.3 [4]. *There is a* CREW *computing logical "or" of n variables in* $\hat{\varphi}(\hat{n})$ *steps.*

There is a gap between values $\varphi(n) \approx \log_{2.618} n$ and $\log_4 n$ denoting the upper and lower bounds of time complexity of the logical "or" of n variables. In this paper, we prove that the algorithm invented in [4] to prove Theorem 1.3 is the optimal one, that is, the time required to compute the logical "or" of n variables is at least $\varphi(n)$ (Theorem 3.9). We get this as a corollary of a more general result. Let n_f be the number of inputs for which function f has value 1. Let $n_f = 2^{i_f} \cdot u$, where u is odd. Then the time needed to compute f on CREW PRAMs is not smaller than $\varphi(n - i_f)$ (Theorem 3.7). This conclusion is quite surprising, since we estimate $CREW(f)$ in terms of the number of inputs $\mathbf{x}$ such that $f(\mathbf{x}) = 1$. This approach has no immediate intuitive motivation, but in many cases it provides better lower bounds than the previous method, based on a very intuitive notion of variables *affecting* a cell at a given time. To prove Theorem

3.7. we connect each state of a cell or processor of a CREW PRAM with a set of Boolean expressions. This state is reached during computation exactly when one of the mentioned Boolean expressions holds for the given input. Then we show that the number of inputs satisfying any of these Boolean expressions is congruent to 0 modulo a number of the form 2^j. Number j decreases after each step. It turns out that after t steps $j = n - F_{2t+1}$. Our main result easily follows from this technical fact.

The methods introduced in this paper have been further developed to get the exact lower time bound for PARITY (Kutyłowski and Reischuk [7]). Reischuk ([9]) noticed that our proofs still work for nondeterministic and probabilistic machines. This is particularly interesting, since no previous proof technique could be used for such machines. Recently, Dietzfelbinger [5] has shown that there is a common structure behind all our proofs. We may express each Boolean function f as a polynomial with integer coefficients. Then, using the same technique as in this paper, it can be shown that $CREW(f) \geqq \varphi(d)$, where d is the degree of this polynomial. As a simple corollary of this fact, a lower bound for symmetric functions was obtained. Perhaps some further results along these lines are possible, since we have obtained a complete representation of CREW PRAM states in terms of Boolean expressions. These Boolean expressions are built by using a few simple operations over literals $x_1, \neg x_1, x_2, \neg x_2, \cdots, x_n, \neg x_n$. So when a Boolean function f over $x_1, \cdots, x_n$ is given, we should express f by applying these operations to literals $x_1, \neg x_1, x_2, \neg x_2, \cdots, x_n, \neg x_n$. Such a representation of f corresponds to a CREW PRAM computing function f. Its time complexity depends on the depth of the representation. The only problem is that one of the mentioned operations is not quite standard and we do not know how to use it efficiently.

2. Boolean PRAMs. For technical reasons, we introduce Boolean PRAMs, a slight modification of (CREW) PRAMs. Boolean PRAMs have a nice property that each state s of a processor (or a cell) is described by a Boolean formula, say, f_s. Formula f_s represents the full knowledge of the processor (of the cell) in state s. Moreover, state s is reached during the computation on input $\mathbf{w}$ if and only if the expression $f_s(\mathbf{w})$ has value 1. More precisely, if $s_1, s_2, \cdots, s_l$ are all possible states of a processor (a cell) P at time point t, then during the computation on input $\mathbf{w}$, P reaches state s_i at time point t if and only if $f_{s_i}(\mathbf{w})$ has value 1 (hence $f_{s_1}, f_{s_2}, \cdots, f_{s_l}$ are mutually exclusive and $f_{s_1} \vee f_{s_2} \vee \cdots \vee f_{s_l} \equiv 1$). It is convenient to identify each state with the formula describing it and we shall always do so.

Now let us explain how the knowledge (hence also the states) of the processors and cells change during a computation of a Boolean PRAM. If a processor P being in state f reads a cell being in state g, then afterwards P knows that $f \wedge g$ holds for the current input. Hence the knowledge of P after the reading is $f \wedge g$, so the state of P should be $f \wedge g$. Similarly, if a processor in state f writes into cell C in state g, then assuming that all knowledge of the processor is appended to the previous content of the cell, the new knowledge of the cell is equal to $f \wedge g$. The only remaining case is when no processor writes into a cell. Suppose that $\{f_1, \cdots, f_l\}$ is the set of all states f such that there is a processor P_f that, in state f during step t, writes into cell C. Then $f_1, \cdots, f_l$ are mutually exclusive by the exclusive write assumption. If, during some computation, no processor writes into C, then nevertheless the knowledge of cell C increases. Indeed, after the writing phase, cell C knows that for the current input the expressions $f_1, \cdots, f_l$ have value 0. Hence $\neg(f_1 \vee \cdots \vee f_l)$ holds. If the previous knowledge of C was g then the new knowledge of C after the writing phase is $g \wedge \neg(f_1 \vee \cdots \vee f_l)$. Hence if the states are to express the full knowledge of the cells, we

must assume that in this particular case the state of cell C changes itself into $g \wedge \neg(f_1 \vee \cdots \vee f_l)$. All this leads to the following definition of Boolean PRAMs.

DEFINITION 2.1. Each Boolean PRAM B is a PRAM for which there is no "computation phase" for the processors and with the property that, if no processor writes into a memory cell, then it changes its content itself. If B obtains n bits on input, then each processor state and each state of a memory cell is a Boolean expression over $x_1, x_2, \cdots, x_n$ (x_i stands for the ith input bit). Let $C_1, C_2, \cdots$ be the cells of B. The states of the processors and cells change as follows:

(i) Initially, all states are equal to 1, except the states of C_i's for $i \leqq n$. The state of C_i ($i \leqq n$) is x_i if the ith bit of the input is 1 and $\neg x_i$ otherwise.

(ii) If processor P in state f reads (writes into) a cell C being in state g, then the state of P (of C) changes to $f \wedge g$.

(iii) Suppose that $\{f_1, \cdots, f_l\}$ is the set of states f for which there is a processor P_f that in state f writes into cell C during step t. If, during step t, no processor writes into cell C that was in state g, then the state of C changes to $g \wedge \neg(f_1 \vee \cdots \vee f_l)$.

From the definition, we get the following properties.

LEMMA 2.2. *Suppose P is a processor or a cell of a Boolean* PRAM *and $t \in \mathcal{N}$. Let $f_1, f_2, \cdots, f_l$ be all possible states of P at step t. Then*

(i) *for each $\mathbf{x} \in \{0, 1\}^n$, exactly one of $f_1(\mathbf{x}), f_2(\mathbf{x}), \cdots, f_l(\mathbf{x})$ holds,*

(ii) *for input $\mathbf{x} \in \{0, 1\}^n$, P reaches state f_i at moment t if and only if $f_i(x)$* holds.

Proof. By the definition, the lemma holds for $t = 0$. Assume that it holds for $t-1$. Let P be a processor and $f_1, \cdots, f_l$ be the possible states of P after step $t-1$. Suppose that in state f_j processor P reads cell C_{m_j} and $g_{j,1}, g_{j,2}, \cdots, g_{j,t(j)}$ are the possible states of C_{m_j} after step $t-1$. Then after the reading, the possible states of P are $f_1 \wedge g_{1,1}$, $f_1 \wedge g_{1,2}, \cdots, f_1 \wedge g_{1,t(1)}, f_2 \wedge g_{2,1}, \cdots, f_2 \wedge g_{2,t(2)}, \cdots, f_l \wedge g_{l,1}, \cdots, f_l \wedge g_{l,t(l)}$. If $\mathbf{x} \in \{0, 1\}^n$, then by the induction hypothesis $f_i(\mathbf{x})$ holds for exactly one i. Similarly, $g_{i,j}$ holds for exactly one j. Hence there is exactly one pair (i, j) such that $f_i(\mathbf{x}) \wedge g_{i,j}(\mathbf{x})$ holds. To prove point (ii), note that P reaches state $f_i \wedge g_{i,j}$ on input $\mathbf{x}$ if and only if, after step $t-1$, processor P is in state f_i and cell C_{m_j} is in state $g_{i,j}$. By the induction hypothesis, it happens if and only if $f_i(\mathbf{x})$ holds and $g_{i,j}(\mathbf{x})$ holds, that is, $f_i(\mathbf{x}) \wedge g_{i,j}(\mathbf{x}) = 1$.

Now let P be a cell and $g_1, \cdots, g_s$ be the possible states of P after step $t-1$. Suppose that $P_{m_1}, P_{m_2}, \cdots, P_{m_l}$ are the processors that may write into cell P during step t and that P_{m_j} writes while being in state f_j (the processors on this list may repeat, but the states $f_1, \cdots, f_l$ are mutually exclusive). Then the possible states of P after the writing are $f_1 \wedge g_1$, $f_1 \wedge g_2, \cdots, f_1 \wedge g_s$, $f_2 \wedge g_1, \cdots, f_2 \wedge g_s, \cdots$, $f_l \wedge g_1, \cdots, f_l \wedge g_s$ and $g_1 \wedge \neg(f_1 \vee \cdots \vee f_l)$, $g_2 \wedge \neg(f_1 \vee \cdots \vee f_l)$, $\cdots$, $g_s \wedge \neg(f_1 \vee \cdots \vee f_l)$. Let $\mathbf{x} \in \{0, 1\}^n$. Then by the induction hypothesis, there is exactly one i such that $g_i(\mathbf{x})$ holds. Hence only $f_1 \wedge g_i$, $f_2 \wedge g_i, \cdots, f_l \wedge g_i$ and $g_i \wedge \neg(f_1 \vee \cdots \vee f_l)$ can be satisfied for $\mathbf{x}$. Since $f_1, \cdots, f_l, \neg(f_1 \vee \cdots \vee f_l)$ are mutually exclusive, exactly one of these expressions holds for $\mathbf{x}$. Hence point (i) holds. If P is a cell, we can show point (ii), as we did for processors above. $\square$

We shall see below that there is a tight connection between Boolean and regular PRAMs.

DEFINITION 2.3. A Boolean PRAM B simulates a CREW PRAM Q if:

(i) While given the same input, the corresponding processors of B and Q read and write exactly in the same memory cells at the same moments.

(ii) There is a function *Sim* with the following property. Let P be a processor (a cell) of B, g a possible state of P at moment t. If P is in state g at step t on input $\mathbf{x}$, then the corresponding processor (cell) of Q is in state $Sim(P, t, g)$ at step t on input $\mathbf{x}$.

LEMMA 2.4. *For each* CREW PRAM Q, *there is a Boolean* PRAM B *simulating* Q.

Proof. At the initial moment, machine B is determined by Definition 2.1(i). $Sim(P, 0, 1)$ is the initial state of processor P, $Sim(C_i, 0, x_i) = 1$, and $Sim(C_i, 0, \neg x_i) = 0$ for $i \leqq n$. For $i > n$, $Sim(C_i, 0, 1)$ is the initial content of cell C_i.

Assume now that machine B has been described up to step t and values of $Sim(\cdots, s, \cdots)$ have been defined for $s \leqq t$. We define B and Sim for $t+1$. Let f be one of the possible states of processor P of B after step t. Let $Sim(P, t, f) = q$. Suppose that processor P of machine Q in state q decides to read from cell C. Processor P of B in state f does the same. Suppose that $g_1, g_2, \cdots, g_u$ are the possible states of cell C of machine B after step t. After reading, processor P changes its state to one of the states $f \wedge g_1, f \wedge g_2, \cdots, f \wedge g_u$. Define $Sim(P, t+1, f \wedge g_i)$ to be the state reached by processor P of machine Q after reading cell C containing $Sim(C, t, g_i)$, if the previous state of P was $Sim(P, t, f)$. Note that the above definition is unambiguous. Indeed, suppose that $f \wedge g_j \equiv f' \wedge g'$. But f' is also a state of processor P of machine B after step t, so by Lemma 2.2, either $f \equiv f'$ or f and f' are mutually exclusive. Hence $f \equiv f'$. Then g' must be a state of cell C after step t. Again by Lemma 2.2, $g \equiv g'$.

Now consider the phase of writing. Let h be one of the possible states of a processor P of B. Then we look at processor P of Q in state $Sim(P, t+1, h)$. It writes to some cell, say C', or does not write at all. Processor P of machine B in state h at step $t+1$ does the same. Then the new state of cell C' is $h \wedge g$, where g is the previous state of C'. Define $Sim(C', t+1, h \wedge g)$ to be the content of cell C' of machine Q that is reached when processor P, being in state $Sim(P, t+1, h)$, writes into cell C' containing $Sim(C', t, g)$. As above, this definition is unambiguous.

It remains to determine Sim for cell C of machine B when no processor writes into it. Let $h = g \wedge \neg(f_1 \vee f_2 \vee \cdots \vee f_u)$ be the new state of C. We set $Sim(C, t+1, h) = Sim(C, t, g)$, since the content of cell C of machine Q does not change during this step.

It follows immediately from the construction that machine Q is simulated by B. □

3. The time bounds. In this section we look more closely at the Boolean expressions serving as the states of Boolean PRAMs.

DEFINITION 3.1. Let M be a Boolean PRAM. Then $Cell(M, i)$ is the set of the states of the cells of M that can be reached at step i. Similarly, $Proc(M, i)$ is the set of all possible states of processors at step i.

DEFINITION 3.2. Let numbers K_t, L_t be defined inductively as follows: $K_0 = 0$, $L_0 = 1$, $K_{t+1} = K_t + L_t$, $L_{t+1} = K_{t+1} + L_t$.

Clearly, K_t, L_t are Fibonacci numbers: $K_t = F_{2t}$, $L_t = F_{2t+1}$.

DEFINITION 3.3. If f is a Boolean expression of n variables, then,

$$A_f = \{\mathbf{x} \in \{0, 1\}^n : f(\mathbf{x}) \text{ holds}\}.$$

DEFINITION 3.4. Suppose $f \in \bigcup_i Cell(M, i) \cup \bigcup_i Proc(M, i)$. We define a coefficient $r(f) \in \mathcal{N}$ as follows. Let j be the minimal number such that $f \in Cell(M, j) \cup Proc(M, j)$. Then $r(f) = K_j$ if $f \in Proc(M, j)$ and $r(f) = L_j$ if $f \in Cell(M, j) \setminus Proc(M, j)$.

Now we are ready to formulate a lemma which is the key to all our results.

LEMMA 3.5. *Let* $f_1, f_2, \cdots, f_s$ *be Boolean expressions from* $\bigcup_i Cell(M, i) \cup \bigcup_i Proc(M, i)$. *Then*

$$|A_{f_1} \cap A_{f_2} \cap \cdots \cap A_{f_s}| \equiv 0 \bmod 2^{n-(r(f_1)+r(f_2)+\cdots+r(f_s))}. \tag{3.1}$$

Proof. The proof is by induction on $l = \max\{r(f_1), r(f_2), \cdots, r(f_s)\}$. If $l = 0$, then each $f_i \in Proc(M, 0)$, so $A_{f_i} = \{0, 1\}^n$. Then $|A_{f_1} \cap A_{f_2} \cap \cdots \cap A_{f_s}| = 2^n$ and the lemma

holds. Assume that $l=1$. Then some f_i's belong to $Cell(M, 0)$, some to $Proc(M, 1)$, and the rest to $Proc(M, 0)$. So each f_i has one of the forms: 1, x_j, or $\neg x_j$ for some j. Hence, without loss of generality, we may assume that $A_{f_i}=\{\mathbf{x}\in\{0,1\}^n: x_{j_i}=c_i\}$ for $i\leqq u$ $(c_i\in\{0,1\})$ and $A_{f_i}=\{0,1\}^n$ for $i>u$. Then

$$A_{f_1}\cap\cdots\cap A_{f_s}=\{\mathbf{x}\in\{0,1\}^n: x_{j_1}=c_1, x_{j_2}=c_2,\cdots, x_{j_u}=c_u\}.$$

The above set is empty if there are two contradictory conditions: $x_{j_{i_1}}=0$, $x_{j_{i_2}}=1$, and $j_{i_1}=j_{i_2}$. In this case the lemma holds, since $|A_{f_1}\cap\cdots\cap A_{f_s}|=0$. Otherwise, $|A_{f_1}\cap\cdots\cap A_{f_s}|=2^{n-p}$ where p is the number of different conditions among $x_{j_1}=c_1, x_{j_2}=c_2,\cdots, x_{j_u}=c_u$. But $p\leqq u=\sum_{i=1}^u r(f_i)\leqq\sum_{i=1}^s r(f_i)$. So

$$2^{n-(r(f_1)+r(f_2)+\cdots+r(f_s))}\,|\,2^{n-p},$$

and the lemma holds.

Assume now that $l>1$. We prove (3.1) by induction on the number of f_i's such that $r(f_i)=l$. Suppose that there are k states f_i such that $r(f_i)=l$. If $k=0$, then $\max\{r(f_1),\cdots,r(f_s)\}<l$, and by the induction hypothesis on l, there is nothing to show. Thus it remains to prove that (3.1) holds for $k\geqq 1$ if it holds for $k-1$. Without loss of generality, we assume that $r(f_1)=l$. Put $A_{f_2}\cap A_{f_3}\cap\cdots\cap A_{f_s}=Rest$ and $r(f_2)+r(f_3)+\cdots+r(f_s)=r$.

Case 1. $r(f_1)=K_t$.

Then $f_1\in Proc(M, t)$ and $f_1=g\wedge h$ for some $g\in Proc(M, t-1)$, $h\in Cell(M, t-1)$. So $A_{f_1}=A_g\cap A_h$. Note that $r(g)\leqq K_{t-1}$, $r(h)\leqq L_{t-1}$. So $r(g), r(h)<K_t=r(f_1)$. Among functions $g, h, f_2,\cdots,f_s$, there are only $k-1$ functions f such that $r(f)=l$. So, by the induction hypothesis,

$$|A_g\cap A_h\cap Rest|\equiv 0 \bmod 2^{n-(r(g)+r(h)+r)}.$$

But $r(g)+r(h)\leqq K_{t-1}+L_{t-1}=K_t=r(f_1)$. So

$$|A_{f_1}\cap Rest|=|A_g\cap A_h\cap Rest|\equiv 0 \bmod 2^{n-(r(f_1)+r)},$$

as required.

Case 2. $r(f_1)=L_t$.

Then $f_1\in Cell(M, t)$. There are two possibilities. Either $f_1=g\wedge h$ for some $g\in Cell(M, t-1)$, $h\in Proc(M, t)$ or $f_1=g\wedge\neg(h_1\vee h_2\vee\cdots\vee h_u)$ for some $g\in Cell(M, t-1)$ and $h_1, h_2,\cdots,h_u\in Proc(M, t)$ such that the sets $A_{h_1}, A_{h_2},\cdots,A_{h_u}$ are disjoint. In the first case, we proceed as in Case 1. So assume that $f_1=g\wedge\neg(h_1\vee h_2\vee\cdots\vee h_u)$. Then

$$\begin{aligned}A_{f_1}\cap Rest&=A_g\cap\overline{(A_{h_1}\cup A_{h_2}\cup\cdots\cup A_{h_u})}\cap Rest\\&=(A_g\cap Rest)\backslash(A_g\cap Rest\cap A_{h_1})\backslash(A_g\cap Rest\cap A_{h_2})\backslash\cdots\backslash(A_g\cap Rest\cap A_{h_u}).\end{aligned}$$

Since $A_{h_1}, A_{h_2},\cdots,A_{h_u}$ are disjoint, so are $(A_g\cap Rest\cap A_{h_1})$, $(A_g\cap Rest\cap A_{h_2}),\cdots,(A_g\cap Rest\cap A_{h_u})$. So

$$|A_{f_1}\cap Rest|=|A_g\cap Rest|-|A_g\cap Rest\cap A_{h_1}|-\cdots-|A_g\cap Rest\cap A_{h_u}|.$$

We show that each one of the numbers on the right side is congruent to 0 modulo $2^{n-(r(f_1)+r)}$. Indeed, $r(g)\leqq L_{t-1}<L_t=r(f_1)$. Then, among functions $g, f_2,\cdots,f_s$, there are only $k-1$ functions f such that $r(f)=l$. So, by the induction hypothesis,

$$|A_g\cap Rest|\equiv 0 \bmod 2^{n-(r(g)+r)}.$$

But $r(g)<r(f_1)$, so

$$2^{n-(r(f_1)+r)}\,|\,2^{n-(r(g)+r)}$$

and

$$|A_g\cap Rest|\equiv 0 \bmod 2^{n-(r(f_1)+r)}.$$

Now consider $|A_g \cap Rest \cap A_{h_i}|$. Since $r(g) \leqq L_{t-1}$, $r(h_i) \leqq K_t$, so $r(g), r(h_i) < L_t = r(f_i)$, and by the induction hypothesis on k,

$$|A_g \cap Rest \cap A_{h_i}| \equiv 0 \bmod 2^{n-(r(g)+r(h_i)+r)}.$$

Note that $r(g)+r(h_i) \leqq L_{t-1}+K_t = L_t = r(f_1)$, so

$$|A_g \cap Rest \cap A_{h_i}| \equiv 0 \bmod 2^{n-(r(f_1)+r)},$$

as required. □

By Lemma 3.5, we immediately get Corollary 3.6.

COROLLARY 3.6. *If $f \in Cell(M, t)$, then*

$$|A_f| \equiv 0 \bmod 2^{n-L_t}.$$

Now we are ready to prove our main result.

THEOREM 3.7. *Suppose f is a Boolean function of n variables and $|A_f| = 2^{i_f} \cdot u$ where u is odd. Then $CREW(f) \geqq \varphi(n-i_f)$.*

Proof. Suppose a machine Q computes function f and the result is always given in a cell C at step t. Consider a Boolean PRAM B simulating Q. Let $g_1, g_2, \cdots, g_u$ be all states g of B such that $Sim(C, t, g) = 1$. Then $A_f = A_{g_1} \cup A_{g_2} \cup \cdots \cup A_{g_u}$ and the sets $A_{g_1}, A_{g_2}, \cdots, A_{g_u}$ are disjoint. By Corollary 3.6, for each $i \leqq u$, we have:

$$|A_{g_i}| \equiv 0 \bmod 2^{n-L_t}.$$

So

$$|A_f| \equiv 0 \bmod 2^{n-L_t}$$

and 2^{n-L_t} divides $|A_f|$. But $|A_f| = 2^{i_f} \cdot u$. So $2^{n-L_t} \mid 2^{i_f}$, that is, $n - L_t \leqq i_f$. Then $n - i_f \leqq L_t = F_{2t+1}$. □

COROLLARY 3.8. *Each* CREW PRAM *computing the logical "or" of n variables requires at least $\varphi(n)$ steps.*

Proof. If f is the logical "or" of n variables, then $|A_f| = 2^n - 1$. So $i_f = 0$. Then we apply Theorem 3.7. □

Cook, Dwork, and Reischuk presented, in [4], an algorithm computing the logical "or" of n variables in $\varphi(n)$ steps. So we get the following result.

THEOREM 3.9. *The optimal* CREW PRAM *computing the logical "or" of n variables makes $\varphi(n)$ steps.*

It is known that the algorithm of Cook, Dwork, and Reischuk can be generalized to the following form.

PROPOSITION 3.10. *For every Boolean function of n variables*

$$CREW(f) \leqq 1 + \varphi(n).$$

Proof. For each possible input string $\mathbf{w}$, there is a cluster of n processors, say $P_{\mathbf{w},0}, P_{\mathbf{w},1}, \cdots, P_{\mathbf{w},n-1}$, and n cells, say $C_{\mathbf{w},0}, \cdots, C_{\mathbf{w},n-1}$, checking if the current input equals $\mathbf{w}$. We describe in detail one such cluster. To check if $\mathbf{w} = x_0, x_1, \cdots, x_{n-1}$, it suffices to compute $s_0 \vee s_1 \vee \cdots \vee s_{n-1}$ where

$$s_i = \begin{cases} x_i & \text{if } w_i = 0, \\ 1 - x_i & \text{if } w_i = 1. \end{cases}$$

During the first step, for $i < n$, processor $P_{\mathbf{w},i}$ reads input x_i, computes s_i, and writes s_i into cell $C_{\mathbf{w},i}$. Starting from the second step, the algorithm of Cook, Dwork, and

Reischuk is used. For each step t, there are numbers p_t, c_t such that after performing step t, for $i<n$:

- processor $P_{\mathbf{w},i}$ knows $s_i \vee \cdots \vee s_{i+p_t-1}$,
- cell $C_{\mathbf{w},i}$ stores $s_i \vee \cdots \vee s_{i+c_t-1}$

(for $j>n$, by s_j we mean $s_{j'}$, where $j'=j \bmod n$). Then, during step $t+1$ the following happens:

- for $i<n$, processor $P_{\mathbf{w},i}$ reads from cell $C_{\mathbf{w},j}$, where $j=i+p_t \bmod n$ (therefore $P_{\mathbf{w},i}$ can compute $s_i \vee \cdots \vee s_{i+p_t+c_t-1}$).
- for $i<n$, if $s_i \vee \cdots \vee s_{i+p_t+c_t-1}=1$, then processor $P_{\mathbf{w},i}$ writes a 1 into cell $C_{\mathbf{w},j}$, where $j=i-c_t \bmod n$. Afterwards, cell $C_{\mathbf{w},j}$ stores $s_j \vee \cdots \vee s_{j+c_t+(p_t+c_t)-1}$ (even if $P_{\mathbf{w},i}$ does not write).

It follows that $p_{i+1}=p_i+c_i$ and $c_{i+1}=c_i+p_{i+1}$, i.e., $p_i=F_{2i-1}$. There are t steps, where $t=\min\{j: F_{2j-1}\geqq n\}=\varphi(n)+1$. Then after the reading of step t, processor $P_{\mathbf{w},1}$ knows $s_0 \vee \cdots \vee s_{n-1}$. If it is 0, then $P_{\mathbf{w},1}$ writes $f(\mathbf{w})$ into the output cell. □

The above algorithm uses $n \cdot 2^n$ processors and cells. This number can be easily reduced to $n \cdot k \leqq n \cdot 2^{n-1}$, where

$$k=\min(|\{\mathbf{x}: f(\mathbf{x})=1\}|, |\{\mathbf{x}: f(\mathbf{x})=0\}|).$$

Indeed, if, for instance, $k=|\{\mathbf{x}: f(\mathbf{x})=1\}|$, then the output cell is initialized with 0 and we take only the clusters corresponding to strings $\mathbf{w}$ such that $f(\mathbf{w})=1$.

COROLLARY 3.11. *If the number of inputs* $\mathbf{x}$ *for which* $f(\mathbf{x})=1$ *is odd, then*

$$\varphi(n) \leqq CREW(f) \leqq \varphi(n)+1.$$

Proof. The proof is immediate by Theorem 3.7 and Proposition 3.10. □

Theorem 3.9 can be used to estimate time complexity of some other important functions. For instance, recall that the threshold function T_n^k is defined as follows:

$$T_n^k(x_1, \cdots, x_n)=1 \quad \text{iff } \sum x_i \geqq k.$$

One of the important cases of the threshold functions is the majority function:

$$MAJORITY_n = T_n^{n/2}.$$

By Theorem 1.2, we get only

$$CREW(MAJORITY_n) \geqq \log_4 \left\lceil \frac{n}{2} \right\rceil \quad \left(\approx \log_4 n - \frac{1}{2}\right).$$

Now we get the following easy corollary of Theorem 3.9.

COROLLARY 3.12. *For each* n *and* $k \leqq n$,

$$CREW(T_n^k) \geqq \varphi\left(\frac{n}{2}\right)-1 \quad (\approx \log_{2.618} n).$$

Proof. Since

$$T_n^k(x_1, \cdots, x_n)=1 \quad \text{iff } T_n^{n-k}(1-x_1, \cdots, 1-x_n)=0,$$

we have

$$CREW(T_n^k)=CREW(T_n^{n-k})$$

and it suffices to prove the lemma under the assumption that $k \leqq (n/2)$.

Let $k \leqq m$. We get the following algorithm for computing the logical "or" of m variables.

Step 1. Write $k-1$ ones and $m-k+1$ zeros into cells $C_m, C_{m+1}, \cdots, C_{2m}$.

Step 2. Start a machine computing T^k_{2m} treating the content of the cells $C_1, \cdots, C_{2m}$ as the input string.

Clearly, for a given input string $x_1, x_2, \cdots, x_m$:

$$x_1 \vee x_2 \vee \cdots \vee x_m = 0 \quad \text{iff} \quad T^k_{2m}(x_1, \cdots, x_m, c_{m+1}, \cdots, c_{2m}) = 0,$$

where c_i is the content of the cell C_i written during the first step. So the algorithm yields the correct answer. For Step 1, only one PRAM step is required (provided there are m processors). For Step 2, we need $CREW(T^k_{2m})$ PRAM steps. So, by Theorem 3.9,

$$CREW(T^k_{2m}) + 1 \geqq \varphi(m).$$

Now we get the desired inequality by substituting $2m$ by n. □

It was proved in [4] that sorting n arbitrary keys requires time between t and $t+5$, where t is the time required for the logical "or" of n variables. So we get the following interesting corollary.

COROLLARY 3.13. *The optimal* PRAM *sorting n keys requires T steps where*

$$\varphi(n) \leqq T \leqq \varphi(n) + 5.$$

4. Final remarks. The results of [4] and this paper provide two different methods of getting lower bounds of the time complexity of Boolean functions. For the logical "or" our method is more precise, but there are many functions (for instance the PARITY function: $\sum x_i \bmod 2$) for which the opposite is true. Hence these two methods are incomparable. In fact, for PARITY we can get almost the same lower bound as for the logical "or" (see [7], [5]).

In order to get Proposition 3.10 we have used, in the worst case, $n \cdot 2^{n-1}$ processors. Hence, from the practical point of view, this method is not interesting at all. The same applies to Corollary 3.13 (compare [4]). The assumption about the exponential number of processors in Proposition 3.10 cannot be dropped, since almost all Boolean functions of n arguments require time $\log n - \log \log p(n) + \Omega(1)$ on CRCW PRAMs with $p(n)$ processors ([1]). On the other hand, the algorithm presented in [4] computing the logical "or" of n variables uses only n processors and n memory cells. So it is both the quickest and a realistic algorithm.

Acknowledgment. I thank Rüdiger Reischuk for presenting me with the considered problems.

Note added in proof. After the article was completed, references [5] and [7] were combined and published as [11].

REFERENCES

[1] P. BEAME AND J. HASTAD, *Optimal bounds for decision problems on the* CRCW PRAM, in Proc. 19th ACM Symposium on Theory of Computing, New York, 1987, pp. 83–93.

[2] A. BORODIN AND J. E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[3] S. BUBLITZ, U. SCHÜRFELD, I. WEGENER, AND B. VOIGT, *Properties of complexity measures for* PRAMs *and* WRAMs, Theoret. Comput. Sci., 48 (1986), pp. 53–73.

[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] M. DIETZFELBINGER, *The structure underlying Kutyłowski's exact lower bounds for* CREW PRAMs, preliminary report, Universität-GH Paderborn, Fachbereich Mathematik-Informatik, Paderborn, Federal Republic of Germany, 1990.

[6] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th ACM Symposium on Theory of Computing, San Diego, CA, 1978, pp. 114–118.

[7] M. KUTYŁOWSKI AND R. REISCHUK, *Evaluating Boolean formulas on parallel machines without simultaneous writes*, Tech. Report, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, Darmstadt, Federal Republic of Germany, 1990.

[8] I. PARBERRY AND P. YUAN YAN, *Improved upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 20 (1991), pp. 88–99.

[9] R. REISCHUK, *Fast evaluation of Boolean formulas by* CREW-PRAMs, Tech. Report TR-89-054, International Computer Science Institute, Berkeley, CA, 1989.

[10] I. WEGENER, *The complexity of Boolean functions*, Wiley–Teubner Series in Computer Science, John Wiley, New York, 1987.

[11] M. DIETZFELBINGER, M. KUTYŁOWSKI, AND R. REISCHUK, *Exact time bounds for computing Boolean functions on* PRAM*s without simultaneous writes*, in Proc. Second ACM Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 125–135.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 834-864, October 1991
© 1991 Society for Industrial and Applied Mathematics
003

EMBEDDING OF GRIDS INTO OPTIMAL HYPERCUBES*

MEE YEE CHAN†

Abstract. This paper addresses the following graph-embedding question: given a grid and the smallest hypercube with at least as many nodes as the grid has grid points, how can grid points be assigned to hypercube nodes (with at most one grid point per node) so as to keep grid-neighbors as near each other as possible in the hypercube? For two-dimensional grids, a simple embedding strategy is introduced which ensures that grid-neighbors are always mapped to hypercube nodes that are within a distance of two edges of each other. Moreover, this embedding algorithm takes time linear to the number of grid points. Extending and adding further concepts, an embedding strategy is formulated for d-dimensional grids which ensures that grid-neighbors are distanced by $O(d)$ edges in the hypercube. This algorithm takes $O(d \times \text{the number of grid points})$ time.

Key words. embedding, hypercubes, grids, dilation, simulation

AMS(MOS) subject classifications. 68M10, 05C10, 94C15

1. Introduction. One of the key features of the hypercube is a rich interconnection structure which permits many important network topologies, such as grids, to be efficiently simulated. A *binary hypercube of dimension n* or *binary n-cube* can be thought of as an undirected graph of 2^n nodes labeled 0 to $2^n - 1$ in binary; two nodes are connected by an edge if and only if their labelings differ in exactly one bit position. To simulate a grid on the hypercube, nodes of the grid must be mapped to hypercube nodes. The question of interest here is: how can we map the nodes of any grid to the nodes of *its optimal hypercube* (the smallest hypercube with at least as many nodes as the grid), on a one-to-one basis, so that *dilation* (the worst case distance between grid-neighbors in the hypercube) is kept to a minimum?

We consider, first of all, two-dimensional grids. A number of researchers have studied this problem [BMS], [BS], [CC], [G], [HJ], [SS], with the following results. Over 61 percent of all two-dimensional grids can be embedded into their optimal hypercubes with a dilation of 1 (i.e., all grid-neighbors are also neighbors in the hypercube) by using binary-reflected Gray codes [SS]. For the other over 38 percent of all two-dimensional grids, which have been proven to need at least dilation 2 [BS], there are the methods proposed in [BMS], [CC], [G], and [HJ]. [BMS], [HJ], and [CC] have shown that a substantial percentage of these grids (over 70 percent of the 38 percent) can be embedded with dilation 2, while [G] shows that all two-dimensional grids can be embedded with dilation 5. The long-time conjecture was that all two-dimensional grids ought to be embeddable in their optimal hypercubes with at most dilation 2 [LS]. This paper introduces a simple embedding strategy which finally confirms this conjecture. In addition, this embedding algorithm is optimal in the sense that it takes $O(\alpha\beta)$ time to embed an $\alpha \times \beta$ grid.

Having obtained the best possible dilation for all two-dimensional grids, the next question is: how can the technique be extended to higher-dimensional grids? Trivial extensions to higher dimensions, unfortunately, tend to cause grids to be mapped to larger-than-optimal-sized hypercubes. The insistence on embedding into optimal hypercubes makes the problem nontrivial and requires a careful embedding strategy.

* Received by the editors April 8, 1988; accepted for publication (in revised form) December 17, 1990. Most of this work was carried out while the author was in the Computer Science Program at the University of Texas at Dallas, Richardson, Texas. The author is also currently associated with James Capel (Far East) Limited, Hong Kong.

† Department of Computer Science, University of Hong Kong, Hong Kong.

The best upper bound up to now for the d-dimensional case is dilation d for a restricted class of grids [BMS]. No lower bound for dilation is known at this time. We give the following result: all d-dimensional grids can be embedded in their optimal hypercubes with dilation at most $4d+1$.

The paper is organized in the following manner. Section 2 introduces the two-dimensional strategy. Section 3 describes the d-dimensional strategy. Section 4 concludes. Brief abstract versions of §§ 2 and 3 appear in [C1] and [C3], respectively.

2. The two-dimensional embedding strategy. Suppose we are given an $\alpha \times \beta$ grid G. Let $\tilde{\alpha} = 2^{\lfloor \log_2 \alpha \rfloor}$ and $\tilde{\beta} = 2^{\lfloor \log_2 \beta \rfloor}$. The binary-reflected Gray code strategy of [SS] already embeds G into its optimal hypercube with dilation 1 when $\alpha\beta > 2\tilde{\alpha}\tilde{\beta}$, or $\alpha = \tilde{\alpha}$ or $\beta = \tilde{\beta}$ (i.e., α or β is a power of two). For this reason, we are only interested in the case where $\alpha\beta \leqq 2\tilde{\alpha}\tilde{\beta}$, $\alpha \neq \tilde{\alpha}$, and $\beta \neq \tilde{\beta}$. For this case, a dilation of at least 2 is necessary for embedding G into its optimal hypercube [BS].

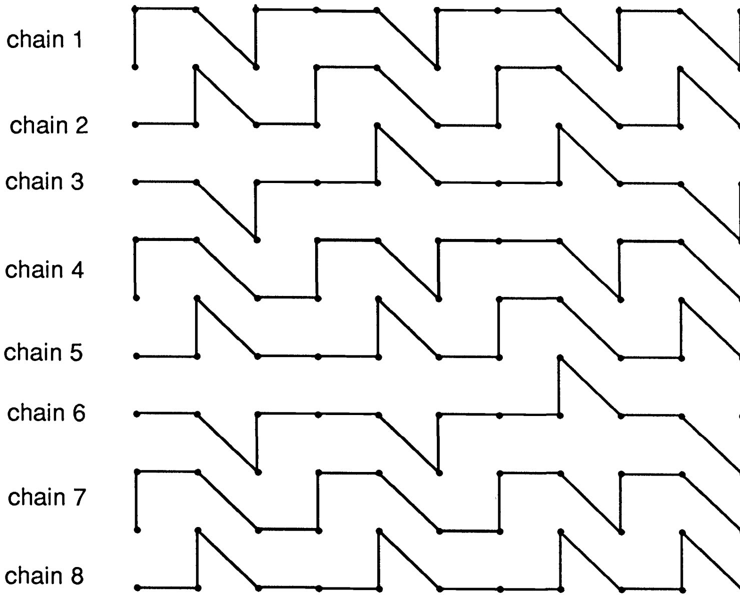
With $\alpha\beta \leqq 2\tilde{\alpha}\tilde{\beta}$, our objective is to label each node of the grid with a unique $(\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor + 1)$-bit binary number, which effectively names the node in the optimal $(\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor + 1)$-cube to which it is mapped. Since we have dilation 2 in mind, we allow the labels for grid-neighbors to differ in at most two bit positions. We will use the 11×11 grid, whose optimal hypercube is the 7-cube, as a running example throughout the rest of this section.

We will determine the label for each node of the grid in two stages: (1) the initial $\lfloor \log_2 \alpha \rfloor$ bits of the label for each node, and then (2) the final $\lfloor \log_2 \beta \rfloor + 1$ bits of the label. To arrive at the first $\lfloor \log_2 \alpha \rfloor$ bits, the nodes of the grid are systematically partitioned into $\tilde{\alpha}$ groups, which we call "chains," of at most $2\tilde{\beta}$ nodes each. The partitioning will be done so that grid-neighbors will be assigned to either the same or adjacent chains. Thus, in the case of an 11×11 grid, we will partition the grid into eight groups, chains 1 through 8, of less than or equal to sixteen nodes each, and if a node is assigned to, say, chain 3, then its grid-neighbors can only belong to chains 2, 3, or 4. Nodes belonging to the same chain will be given the same bits as the first $\lfloor \log_2 \alpha \rfloor$ bits of its label. The first $\lfloor \log_2 \alpha \rfloor$ bits given to nodes of chain i will differ from the first $\lfloor \log_2 \alpha \rfloor$ bits given to nodes of chain $i+1$ by exactly one bit. Hence, the first $\lfloor \log_2 \alpha \rfloor$ bits of the labels assigned to grid-neighbors will differ in at most one bit position. The last $\lfloor \log_2 \beta \rfloor + 1$ bits are determined in two substages. The first substage ensures that (i) the last $\lfloor \log_2 \beta \rfloor + 1$ bits for nodes within the same chain are unique and (ii) the last $\lfloor \log_2 \beta \rfloor + 1$ bits of grid-neighbors differ by no more than two bits; thus, at this point, labels for grid-neighbors may differ by as much as three bits, namely, when their first $\lfloor \log_2 \alpha \rfloor$ bits differ by one bit and their last $\lfloor \log_2 \beta \rfloor + 1$ bits differ by two bits. The second substage makes a small adjustment to the last $\lfloor \log_2 \beta \rfloor + 1$ bits of the labels to force dilation 2.

Both stages rely heavily on a very special *partitioning matrix* $A(\alpha, \beta)$, or simply A, an integer matrix comprised of 1's and 2's having $\tilde{\alpha}$ rows and β columns. As we shall see, with regard to the partitioning of the nodes of grid G into $\tilde{\alpha}$ groups or chains, $a_{i,j}$ (i.e., the element in the ith row, jth column of matrix A) essentially indicates how many nodes from column j of grid G will belong to chain i. Thus, because we wish to partition G into $\tilde{\alpha}$ chains, matrix A has $\tilde{\alpha}$ rows, and because there are β columns in grid G, matrix A has β columns. The actual partitioning scheme based on matrix A will be described later. However, one can get some feeling about the scheme by referring to the example of matrix A for the 11×11 grid given on the next page and the corresponding division of the 11×11 grid into eight chains given in Fig. 2.1(b). For now, let us define matrix A and take note of some of its properties. Matrix A has

$$
\text{CHAIN} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 1 & 2 & 2 & 1 & 2 & 2 & 1 & 2 & 1 \\
2 & 2 & 2 & 2 & 3 & 2 & 2 & 3 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 \\
4 & 5 & 4 & 4 & 5 & 4 & 5 & 5 & 4 & 5 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 6 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
7 & 7 & 6 & 7 & 7 & 6 & 7 & 7 & 7 & 7 & 6 \\
7 & 8 & 7 & 7 & 8 & 7 & 7 & 8 & 7 & 8 & 7 \\
8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8
\end{bmatrix}
$$

(a)



(b)

FIG. 2.1. *An example of CHAIN assignment for an* 11×11 *grid.*

as its first column the vector

$$
\begin{bmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \\ a_{4,1} \\ \vdots \\ a_{\tilde{\alpha},1} \end{bmatrix} = \begin{bmatrix} \lceil \alpha/\tilde{\alpha} \rceil \\ \lfloor \alpha/\tilde{\alpha} \rfloor \\ \lfloor 2\alpha/\tilde{\alpha} \rfloor - \lfloor \alpha/\tilde{\alpha} \rfloor \\ \lfloor 3\alpha/\tilde{\alpha} \rfloor - \lfloor 2\alpha/\tilde{\alpha} \rfloor \\ \vdots \\ \lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-2)\alpha/\tilde{\alpha} \rfloor \end{bmatrix}.
$$

Precisely, for all $1 \leqq i \leqq \tilde{\alpha}$, $a_{i,1} = \lfloor (i-1)\alpha/\tilde{\alpha} \rfloor - \lfloor (i-2)\alpha/\tilde{\alpha} \rfloor$. Hence, for the 11×11

grid, the first column vector is

$$\begin{bmatrix} 2 \\ 1 \\ 1 \\ 2 \\ 1 \\ 1 \\ 2 \\ 1 \end{bmatrix}.$$

Intuitively, this vector is defined in such a way as to evenly space out the 2's amidst the 1's when there are more 1's than 2's (i.e., when α is closer to $\tilde{\alpha}$) and the 1's amidst the 2's when there are more 2's than 1's (i.e., when α is closer to $2\tilde{\alpha}$). Note that the first column vector is defined independent of β. Before we start to prove the properties of A, let us state the following inequalities which will be useful in proving subsequent lemmas.

$$\lfloor d/c \rfloor \leqq \lfloor (b+d)/c \rfloor - \lfloor b/c \rfloor \leqq \lceil d/c \rceil \tag{1}$$

for any integers b, c, and d. (This inequality follows directly from the inequality

$$\lfloor b/c \rfloor + \lfloor d/c \rfloor \leqq \lfloor (b+d)/c \rfloor \leqq \lfloor b/c \rfloor + \lceil d/c \rceil.)$$

$$\lfloor (b+d)/c \rfloor \leqq \lfloor b/c \rfloor + \lceil d/c \rceil \leqq \lceil (b+d)/c \rceil \tag{2}$$

for any integers b, c and d.

We can verify that each element in the first column of A is indeed either 1 or 2.

LEMMA 2.1.1. *For all* $1 \leqq i \leqq \tilde{\alpha}$, $1 \leqq a_{i,1} \leqq 2$.

Proof. From (1) and $\alpha > \tilde{\alpha}$, we have, for all $1 \leqq i \leqq \tilde{\alpha}$,

$$1 \leqq \lfloor \alpha/\tilde{\alpha} \rfloor \leqq a_{i,1} = \lfloor (i-1)\alpha/\tilde{\alpha} \rfloor - \lfloor (i-2)\alpha/\tilde{\alpha} \rfloor \leqq \lceil \alpha/\tilde{\alpha} \rceil \leqq 2. \qquad \square$$

The entire matrix A is based on a cyclic shift of the first column, i.e., for all $1 \leqq i < \tilde{\alpha}$ and $1 \leqq j < \beta$, $a_{i,j} = a_{i+1,j+1}$ and $a_{\tilde{\alpha},j} = a_{1,j+1}$. We have for all $1 \leqq i \leqq \tilde{\alpha}$ and $1 \leqq j \leqq \beta$, $a_{i,j} = a_{(i-j) \bmod \tilde{\alpha}+1,1}$:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,\beta} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,\beta} \\ a_{3,1} & a_{3,2} & \cdots & a_{3,\beta} \\ \vdots & \vdots & & \vdots \\ a_{\tilde{\alpha},1} & a_{\tilde{\alpha},2} & \cdots & a_{\tilde{\alpha},\beta} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{\tilde{\alpha},1} & a_{\tilde{\alpha}-1,1} & \cdots \\ a_{2,1} & a_{1,1} & a_{\tilde{\alpha},1} & \cdots \\ a_{3,1} & a_{2,1} & a_{1,1} & \cdots \\ \vdots & \vdots & \vdots & \\ a_{\tilde{\alpha},1} & a_{\tilde{\alpha}-1,1} & a_{\tilde{\alpha}-2,1} & \cdots \end{bmatrix}.$$

Thus, for the 11×11 grid, matrix A is the following 8×11 matrix:

$$\begin{bmatrix} 2 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 \\ 2 & 1 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 1 & 1 \\ 1 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 1 & 2 \\ 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 2 & 1 & 1 \\ 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 & 2 & 1 \end{bmatrix}.$$

For $k_1 \leqq k_2$, let $ROWSUM(i; k_1, k_2) \equiv \sum_{j=k_1}^{k_2} a_{i,j}$ and let $COLSUM(j; k_1, k_2) \equiv \sum_{i=k_1}^{k_2} a_{i,j}$; for $k_1 > k_2$, $ROWSUM(i; k_1, k_2) \equiv 0$ and $COLSUM(j; k_1, k_2) \equiv 0$.

The cyclic construction and the even distribution of the 2's along each column makes possible the following lemmas about properties of the functions *COLSUM* and *ROWSUM*. To provide some insight as to why we are interested in proving the following two lemmas, remember that $a_{i,j}$ will tell us how many nodes from column j of grid G will belong to chain i. So, to make sure all of the nodes of each column of G are assigned to chains, we need $COLSUM(j; 1, \tilde{\alpha}) = \alpha$ for all $1 \leqq j \leqq \beta$ (Corollary 2.1.1). Also, because we would like each chain to have at most $2\tilde{\beta}$ nodes, we need $ROWSUM(i; 1, \beta) \leqq 2\tilde{\beta}$ for all $1 \leqq i \leqq \tilde{\alpha}$ (see Corollary 2.1.2). In fact, the following two lemmas state stronger results: the $COLSUM(ROWSUM)$ of any column (row) from the first to the kth element differs by no more than one from any other.

LEMMA 2.1.2. *For all* $1 \leqq j \leqq \beta$, $1 \leqq k \leqq \tilde{\alpha}$, $\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq COLSUM(j; 1, k) \leqq \lceil k\alpha/\tilde{\alpha} \rceil$.

Proof. With the cyclic property of matrix A and the telescoping series property of the elements in A, we have

$$COLSUM(j; 1, k)$$

$$= \sum_{i=1}^{k} a_{i,j} = \sum_{i=1}^{k} a_{(i-j) \bmod \tilde{\alpha}+1,1} = \begin{cases} \sum_{i=1}^{k} a_{\tilde{\alpha}+i-j+1,1} & \text{if } 1 \leqq k < j \\ \sum_{i=1}^{j-1} a_{\tilde{\alpha}+i-j+1,1} + \sum_{i=j}^{k} a_{i-j+1,1} & \text{if } 1 \leqq j \leqq k \end{cases}$$

$$= \begin{cases} \lfloor (\tilde{\alpha}-j+k)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-j)\alpha/\tilde{\alpha} \rfloor & \text{if } 1 \leqq k < j \\ \lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-j)\alpha/\tilde{\alpha} \rfloor + \lceil \alpha/\tilde{\alpha} \rceil + \lfloor (k-j)\alpha/\tilde{\alpha} \rfloor & \text{if } 1 \leqq j \leqq k. \end{cases}$$

When $k < j$, from (1), we have

$$\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq \lfloor (\tilde{\alpha}-j+k)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-j)\alpha/\tilde{\alpha} \rfloor \leqq \lceil k\alpha/\tilde{\alpha} \rceil.$$

As for $j \leqq k$, from (2), we have $\alpha \leqq \lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha} \rfloor + \lceil \alpha/\tilde{\alpha} \rceil \leqq \alpha$, and from (1), we have

$$\lfloor (k-\tilde{\alpha})\alpha/\tilde{\alpha} \rfloor \leqq \lfloor (k-j)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-j)\alpha/\tilde{\alpha} \rfloor \leqq \lceil (k-\tilde{\alpha})\alpha/\tilde{\alpha} \rceil.$$

With the above inequalities and (2), we have the following:

$$\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq \alpha + \lfloor (k-\tilde{\alpha})\alpha/\tilde{\alpha} \rfloor \leqq \lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha} \rfloor + \lceil \alpha/\tilde{\alpha} \rceil - \lfloor (\tilde{\alpha}-j)\alpha/\tilde{\alpha} \rfloor + \lfloor (k-j)\alpha/\tilde{\alpha} \rfloor$$
$$\leqq \alpha + \lceil (k-\tilde{\alpha})\alpha/\tilde{\alpha} \rceil \leqq \lceil k\alpha/\tilde{\alpha} \rceil. \qquad \square$$

COROLLARY 2.1.1. *For all* $1 \leqq j \leqq \beta$, $COLSUM(j; 1, \tilde{\alpha}) = \alpha$.

Proof. The proof follows from Lemma 2.1.2. $\square$

LEMMA 2.1.3. *For all* $1 \leqq i \leqq \tilde{\alpha}$, $1 \leqq k \leqq \beta$, $\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq ROWSUM(i; 1, k) \leqq \lceil k\alpha/\tilde{\alpha} \rceil$.

Proof. Let $k = p\tilde{\alpha} + q$ where p, q are integers and $0 \leqq q < \tilde{\alpha}$. With the cyclic property of matrix A and the telescoping series property of the elements in A, since $COLSUM(1; 1, \tilde{\alpha}) = \sum_{i=1}^{\tilde{\alpha}} a_{i,1} = \alpha$ by Corollary 2.1.1, we have

$$ROWSUM(i; 1, k)$$

$$= \sum_{j=1}^{k} a_{i,j} = \sum_{j=1}^{p\tilde{\alpha}+q} a_{(i-j) \bmod \tilde{\alpha}+1,1} = \sum_{j=1}^{p\tilde{\alpha}} a_{(i-j) \bmod \tilde{\alpha}+1,1} + \sum_{j=p\tilde{\alpha}+1}^{p\tilde{\alpha}+q} a_{(i-j) \bmod \tilde{\alpha}+1,1}$$

$$= p \sum_{i=1}^{\tilde{\alpha}} a_{i,1} + \sum_{j=1}^{q} a_{(i-j) \bmod \tilde{\alpha}+1,1} = p\alpha + \sum_{j=1}^{q} a_{(i-j) \bmod \tilde{\alpha}+1,1}$$

$$= \begin{cases} p\alpha + \sum_{j=1}^{q} a_{i-j+1,1} & \text{if } 0 \leqq q < i \leqq \tilde{\alpha} \\ p\alpha + \sum_{j=1}^{i} a_{i-j+1,1} + \sum_{j=i+1}^{q} a_{\tilde{\alpha}+i-j+1,1} & \text{if } 1 \leqq i \leqq q < \tilde{\alpha} \end{cases}$$

$$=\begin{cases} p\alpha+\lfloor (i-1)\alpha/\tilde{\alpha}\rfloor-\lfloor (i-1-q)\alpha/\tilde{\alpha}\rfloor & \text{if } 0\leqq q<i\leqq\tilde{\alpha} \\ p\alpha+\lfloor (i-1)\alpha/\tilde{\alpha}\rfloor+\lceil \alpha/\tilde{\alpha}\rceil+\lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha}\rfloor-\lfloor (\tilde{\alpha}-q+i-1)\alpha/\tilde{\alpha}\rfloor & \\ & \text{if } 1\leqq i\leqq q<\tilde{\alpha}. \end{cases}$$

When $q<i$, from (1),

$$\lfloor k\alpha/\tilde{\alpha}\rfloor=p\alpha+\lfloor q\alpha/\tilde{\alpha}\rfloor\leqq p\alpha+\lfloor (i-1)\alpha/\tilde{\alpha}\rfloor-\lfloor (i-1-q)\alpha/\tilde{\alpha}\rfloor$$
$$\leqq p\alpha+\lceil q\alpha/\tilde{\alpha}\rceil\leqq\lceil k\alpha/\tilde{\alpha}\rceil.$$

As for $i\leqq q$, from (2), we have $\alpha\leqq\lceil \alpha/\tilde{\alpha}\rceil+\lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha}\rfloor\leqq\alpha$, and from (1), we have

$$\lfloor (q-\tilde{\alpha})\alpha/\tilde{\alpha}\rfloor\leqq\lfloor (i-1)\alpha/\tilde{\alpha}\rfloor-\lfloor (\tilde{\alpha}-q+i-1)\alpha/\tilde{\alpha}\rfloor\leqq\lceil (q-\tilde{\alpha})\alpha/\tilde{\alpha}\rceil.$$

From these inequalities and (2), we have the following:

$$\lfloor k\alpha/\tilde{\alpha}\rfloor=p\alpha+\alpha+\lfloor (q-\tilde{\alpha})\alpha/\tilde{\alpha}\rfloor$$
$$\leqq p\alpha+\lfloor (i-1)\alpha/\tilde{\alpha}\rfloor+\lceil \alpha/\tilde{\alpha}\rceil+\lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha}\rfloor-\lfloor (\tilde{\alpha}-q+i-1)\alpha/\tilde{\alpha}\rfloor$$
$$\leqq p\alpha+\alpha+\lceil (q-\tilde{\alpha})\alpha/\tilde{\alpha}\rceil=\lceil k\alpha/\tilde{\alpha}\rceil.$$

□

COROLLARY 2.1.2. *For all* $1\leqq i\leqq\tilde{\alpha}$,

$$ROWSUM(i;1,\beta)\leqq\lceil \beta\alpha/\tilde{\alpha}\rceil\leqq 2\tilde{\beta}.$$

Proof. Follows from Lemma 2.1.3 and $\alpha\beta\leqq 2\tilde{\alpha}\tilde{\beta}$. □

Armed with matrix A, we are now ready to talk about the embedding strategy. To make things somewhat easier to understand, we will actually show first how to embed G into its optimal hypercube with dilation 3, discuss the properties of the dilation-3 embedding, and then modify the embedding to produce a dilation-2 embedding. Henceforth, we use $[x,y]$ to denote the node in row x, column y of the grid G, $1\leqq x\leqq\alpha$ and $1\leqq y\leqq\beta$. The dilation-3 embedding strategy is outlined below. Explanations for each step of the strategy are found in the paragraphs following the outline.

DILATION-3 EMBEDDING STRATEGY.
(1) Construct $A(\alpha,\beta)$.
(2) For each node $[x,y]$ of G, compute $CHAIN[x,y]$, $(1\leqq CHAIN[x,y]\leqq\tilde{\alpha})$, where $CHAIN[x,y]=z$ if and only if $COLSUM(y;1,z-1)<x\leqq COLSUM(y;1,z)$.
(3) For each node $[x,y]$ of G, compute $NUMBER[x,y]$, $(1\leqq NUMBER[x,y]\leqq 2\tilde{\beta})$, where $NUMBER[x,y]=1+ROWSUM(CHAIN[x,y];1,y-1)+\delta[x,y]$ and

$$\delta[x,y]=\begin{cases}1 & \text{if } CHAIN[x+1,y]=CHAIN[x,y]\\ 0 & \text{otherwise.}\end{cases}$$

(4) For each node $[x,y]$ of G, compute $MARK[x,y]$ where

$$MARK[x,y]=CHAIN[x,y]-COLSUM(1;1,CHAIN[x,y])+NUMBER[x,y].$$

(5) Let $GRAY(t,p)$ denote the $((p-1)\bmod 2^t+1)$th element of the t-bit binary-reflected Gray code sequence. (For example, $GRAY(3,4)\equiv 010$ since 010 is the fourth element of (000,001,011,010,110,111,101,100).) Let the first $\lfloor\log_2\alpha\rfloor$ bits of the label given to node $[x,y]$ of G be $GRAY(\lfloor\log_2\alpha\rfloor, CHAIN[x,y])$ and the last $\lceil\log_2\beta\rceil+1$ bits be $GRAY(\lfloor\log_2\beta\rfloor+1, MARK[x,y])$.

In step (2), the nodes of G are partitioned into $\tilde{\alpha}$ chains. $CHAIN[x,y]$ tells us to which chain the node $[x,y]$ belongs. Since $a_{i,j}$ essentially tells us how many nodes

from column j of grid G belong to chain i, there will be either one or two nodes from any column of G belonging to a chain. In particular, if $[x, y]$ is the only node from column y belonging to $CHAIN[x, y]=z$, then $a_{z,y}=1$, $COLSUM(y; 1, z)=x$, $COLSUM(y; 1, z-1)=x-1$, and $COLSUM(y; 1, z+1)\geqq x+1$. Alternatively, if both $[x, y]$ and $[x+1, y]$ belong to $CHAIN[x, y]=CHAIN[x+1, y]=z$, then $a_{z,y}=2$, $COLSUM(y; 1, z)=x+1$, $COLSUM(y; 1, z-1)=x-1$, and $COLSUM(y; 1, z+1)\geqq x+2$.

From the way $CHAIN[x, y]$ is defined and from Corollary 2.1.1, clearly $1\leqq CHAIN[x, y]\leqq \tilde{\alpha}$. Since there are $a_{i,j}$ nodes from column j of grid G belonging to chain i, the number of nodes in chain i is $ROWSUM(i; 1, \beta)$ and $ROWSUM(i; 1, \beta)\leqq 2\tilde{\beta}$ according to Corollary 2.1.2. Figure 2.1(a) gives the 11×11 $CHAIN$ values for the 11×11 grid. Figure 2.1(b) uses line segments to join together nodes belonging to the same chain. The convention used in drawing line segments is the following: two nodes in the same column belonging to the same chain are joined by a line segment, and the topmost node in column j belonging to chain i is joined to the bottommost node in column $j+1$ belonging to the same chain i. The reader can easily verify that in Fig. 2.1, for the 11×11 grid, there are less than or equal to 16 nodes belonging to each of the eight chains, and grid-neighbors are given $CHAIN$ values that differ by at most 1.

Before we proceed further to prove formally that the $CHAIN$ values given to grid-neighbors will differ by at most 1, we shall introduce some lemmas and corollaries which will be useful in the proofs of later results. The following lemma states the relationship between the $COLSUM$s of adjacent columns and is based on the cyclic construction of A.

LEMMA 2.1.4. *$COLSUM(y+1; 1, z)=COLSUM(y; 1, z)-a_{z,y}+a_{1,y+1}$.*

Proof.

$$COLSUM(y+1; 1, z)=\sum_{i=1}^{z} a_{i,y+1}=a_{1,y+1}+\sum_{i=2}^{z} a_{i,y+1}=a_{1,y+1}+\sum_{i=2}^{z} a_{i-1,y}$$

$$=a_{1,y+1}-a_{z,y}+\sum_{i=1}^{z} a_{i,y}=a_{1,y+1}-a_{z,y}+COLSUM(y; 1, z).$$

□

COROLLARY 2.1.3. *If $COLSUM(y; 1, z)=COLSUM(y+1; 1, z)$, then $a_{1,y+1}=a_{z,y}$.*

Proof. The proof directly follows from Lemma 2.1.4. □

COROLLARY 2.1.4. *If $COLSUM(y; 1, z)+1=COLSUM(y+1; 1, z)$, then $a_{1,y+1}=2$ and $a_{z,y}=1$.*

Proof. The proof follows from Lemmas 2.1.4 and 2.1.1. □

With an argument similar to that used in Lemma 2.1.4, we have a similar statement concerning $ROWSUM$s.

LEMMA 2.1.5. *$ROWSUM(z+1; 1, y)=ROWSUM(z; 1, y)+a_{z+1,1}-a_{z,y}$.*

The following observation can be made concerning the topmost node in column j belonging to chain i and the bottommost node in column $j+1$ belonging also to chain i. It will be used extensively in the proofs of later lemmas. Based on this observation, we see that line segments of any chain can only be between horizontal grid-neighbors (i.e., $[x, y]$ and $[x, y+1]$), or vertical grid-neighbors (i.e., $[x, y]$ and $[x+1, y]$), or that they slope down at most one row from $[x, y]$ to $[x+1, y+1]$.

LEMMA 2.1.6. *Let $[x, y]$ be the topmost node in column y belonging to chain z and $[x', y+1]$ be the bottommost node in column $y+1$ belonging also to chain z. Then, $0\leqq x'-x\leqq 1$.*

Proof. Since $[x, y]$ is the topmost node in column y belonging to chain z, $COLSUM(y; 1, z) = x + a_{z,y} - 1$. Since $[x', y+1]$ is the bottommost node in column $y+1$ belonging also to chain z, $COLSUM(y+1; 1, z) = x'$. Furthermore, from Lemma 2.1.4,

$$\begin{aligned} COLSUM(y+1; 1, z) &= COLSUM(y; 1, z) - a_{z,y} + a_{1,y+1} \\ &= x + a_{z,y} - 1 - a_{z,y} + a_{1,y+1} = x - 1 + a_{1,y+1} = x'. \end{aligned}$$

Thus, $x' - x = a_{1,y+1} - 1$. Since $a_{1,y+1} \in \{1, 2\}$, $x' \in \{x, x+1\}$. □

Because of the cyclic construction of matrix A and because matrix A is comprised solely of 1's and 2's (see Lemma 2.1.1), we have the following lemmas concerning *CHAIN* values, i.e., grid-neighbors are either assigned to the same chain or adjacent chains.

LEMMA 2.2.1. *For all* $1 \leqq x < \alpha$ *and* $1 \leqq y \leqq \beta$, $0 \leqq CHAIN[x+1, y] - CHAIN[x, y] \leqq 1$.

Proof. Suppose $CHAIN[x, y] = z$. By definition,

$$COLSUM(y; 1, z-1) < x \leqq COLSUM(y; 1, z).$$

It can be seen that

$$\begin{aligned} COLSUM(y; 1, z-1) < x+1 &\leqq COLSUM(y; 1, z) + 1 \\ &\leqq COLSUM(y; 1, z) + a_{z+1,y} = COLSUM(y; 1, z+1). \end{aligned}$$

Thus, $z \leqq CHAIN[x+1, y] \leqq z+1$. □

LEMMA 2.2.2. *For all* $1 \leqq x \leqq \alpha$ *and* $1 \leqq y < \beta$,

$$-1 \leqq CHAIN[x, y+1] - CHAIN[x, y] \leqq 1.$$

Proof. Suppose $CHAIN[x, y] = z$. By definition,

$$COLSUM(y; 1, z-1) < x \leqq COLSUM(y; 1, z).$$

From Lemma 2.1.4 and Lemma 2.1.1, we have

$$\begin{aligned} COLSUM(y; 1, z-1) &= COLSUM(y+1; 1, z-1) + a_{z-1,y} - a_{1,y+1} \\ &= COLSUM(y+1; 1, z-2) + a_{z-1,y+1} + a_{z-1,y} - a_{1,y+1} \\ &\geqq COLSUM(y+1; 1, z-2). \end{aligned}$$

Similarly,

$$\begin{aligned} COLSUM(y; 1, z) &= COLSUM(y+1; 1, z) + a_{z,y} - a_{1,y+1} \\ &= COLSUM(y+1; 1, z+1) - a_{z+1,y+1} + a_{z,y} - a_{1,y+1} \\ &\leqq COLSUM(y+1; 1, z+1). \end{aligned}$$

Thus,

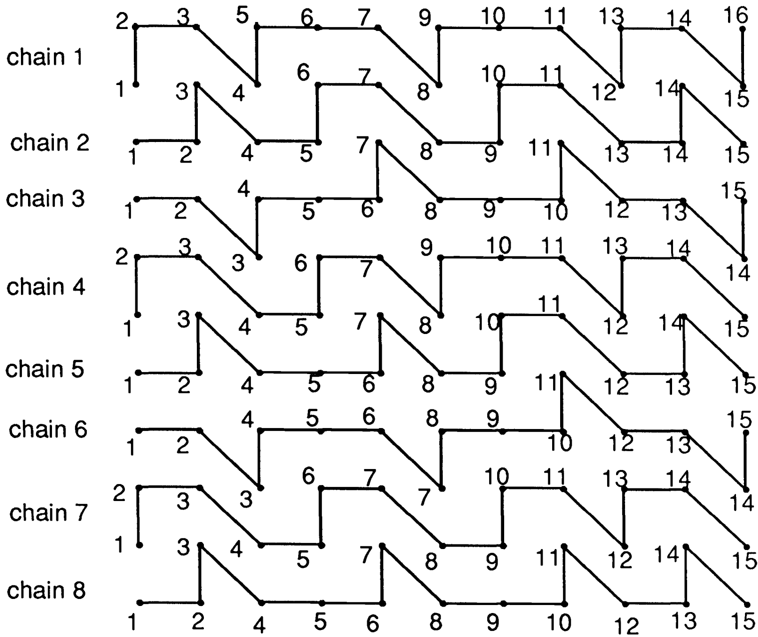
$$COLSUM(y+1; 1, z-2) < x \leqq COLSUM(y+1; 1, z+1)$$

and $z-1 \leqq CHAIN[x, y+1] \leqq z+1$. □

COROLLARY 2.2.1. *The first* $\lfloor \log_2 \alpha \rfloor$ *bits of the labels assigned to grid-neighbors will differ in at most one bit position.*

Proof. Lemmas 2.2.1 and 2.2.2 show that the *CHAIN* values of grid-neighbors differ by at most one. Since the first $\lfloor \log_2 \alpha \rfloor$ bits of the label assigned to $[x, y]$ are $GRAY(\lfloor \log_2 \alpha \rfloor, CHAIN[x, y])$, the first $\lfloor \log_2 \alpha \rfloor$ bits of grid-neighbors will differ in at most one bit position. □

Next, in step (3), the nodes of each chain are numbered uniquely and sequentially, starting at 1 and proceeding along the line segments. $NUMBER[x, y]$ denotes the



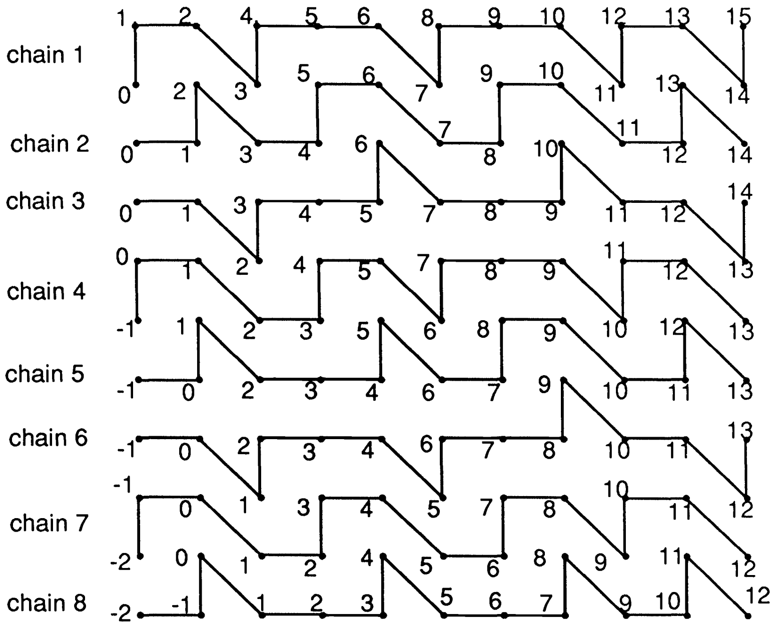


FIG. 2.2. *Example of NUMBER and MARK values for an* 11×11 *grid.*

number given to node $[x, y]$. Figure 2.2(a) shows the *NUMBER* values given to each node of the 11×11 grid. $\delta[x, y]$ is used to account for the case when $[x, y]$ and $[x+1, y]$ belong to the same chain and is defined in such a way that $NUMBER[x, y] = NUMBER[x+1, y]+1$ when $CHAIN[x, y] = CHAIN[x+1, y]$ to be consistent with the convention that the topmost node in column j of a chain is joined to the bottommost node in column $j+1$ of the same chain. For example, in Fig. 2.2(a), consider nodes $[2,4]$ and $[3,4]$: $CHAIN[2,4] = CHAIN[3,4] = 2$, and $NUMBER[2,4] = 6$ and $NUMBER[3,4] = 5$. In general, if $\delta[x, y] = 1$, then $a_{CHAIN[x,y],y} = 2$ and $\delta[x+1, y] = 0$ based on the fact that no three nodes from the same column belong to the same chain.

As it turns out, if we were to assign $GRAY(\lfloor \log_2 \beta \rfloor + 1, NUMBER[x, y])$ as the last $\lfloor \log_2, \beta \rfloor + 1$ bits for the label of node $[x, y]$, we would arrive at a dilation-4 embedding strategy, since it can, in fact, be proved that the numbers assigned to grid-neighbors differ by at most 3. Close study of Fig. 2.2(a) will reveal that the difference of 3 occurs when the grid-neighbors are on different chains. For example, nodes $[5,3]$ and $[5,4]$ are grid-neighbors which belong to adjacent chains whose *NUMBER* values differ by 3. In fact, one can show that the difference is at most 2 for grid-neighbors on the same chain. As it turns out, a simple adjustment or shifting of this numbering scheme on each chain can ensure that the numbers assigned to all grid-neighbors differ by at most 2 and thus achieve dilation 3. In Fig. 2.2(a), note that *by subtracting one* from each of the *NUMBER* values associated with nodes of chain 4, chain 4 can be "synchronized" with chain 3 in the sense that if node $[x+1, y]$ belonged to chain 4 and node $[x, y]$ (the node above $[x+1, y]$) belonged to chain 3, they would then have the same *NUMBER* value. This synchronization allows the *NUMBER* values of nodes $[5,3]$ and $[5,4]$ to no longer differ by 3 but rather by 2. In general, we wish to have every chain synchronized with the one above it. The benefits of synchronization arise when considering horizontal grid-neighbors, $[x, y+1]$ right of $[x, y]$, that belong to different chains: synchronization limits the possible scenarios to those shown in Fig. 2.3. Figure 2.3(a) depicts the only scenario possible for $[x, y]$ belonging to the chain above $[x, y+1]$'s (Case ii in Lemma 2.3.3), while Fig. 2.3(b) depicts cases where $[x, y]$ belongs to the chain below $[x, y+1]$'s (Case iii of Lemma 2.3.3). Thus, in step (4), to achieve synchronization, the numbering for chain z is adjusted or shifted by an offset of $z - COLSUM(1; 1, z)$ to arrive at *MARK* values. Figure 2.2(b) gives the *MARK* values for the 11×11 grid. The reader can easily verify that, for the 11×11 grid, grid-neighbors have *MARK* values which differ by at most 2.

We shall show in Lemma 2.3.2 that indeed we have synchronization with the *MARK* values; in other words, if node $[x+1, y]$ belongs to chain $z+1$ and node $[x, y]$ (above $[x+1, y]$) belongs to chain z, $MARK[x+1, y] = MARK[x, y]$. Since $NUMBER[x, y] = NUMBER[x+1, y]+1$ when $CHAIN[x, y] = CHAIN[x+1, y]$

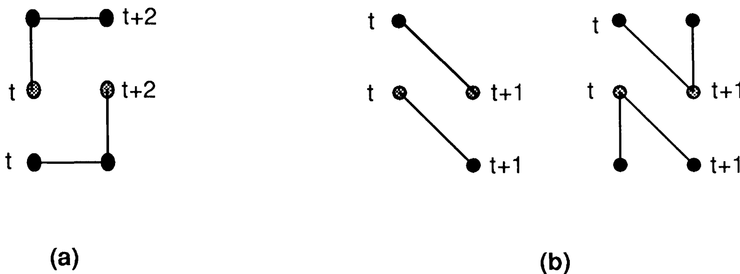


FIG. 2.3. *All possible scenarios for MARK values of horizontal grid-neighbors.*

and *MARK* values are just shifted *NUMBER* values, we also expect $MARK[x, y] = MARK[x+1, y]+1$ when $CHAIN[x, y] = CHAIN[x+1, y]$; this is formally argued in Lemma 2.3.1. So, Lemma 2.3.1 tells us that vertical grid-neighbors $[x, y]$ and $[x+1, y]$ belonging to the same chain will have *MARK* values that differ by 1, and Lemma 2.3.2 tells us that vertical grid-neighbors belonging to different chains will have the same *MARK* value. Hence, Lemmas 2.3.1 and 2.3.2 combined take care of vertical grid-neighbors. Lemma 2.3.3 tackles horizontal grid-neighbors.

Finally, since the nodes of chain z received unique *NUMBER* values ranging from 1 to $ROWSUM(z; 1, \beta)$, the nodes of chain z will receive unique *MARK* values ranging from $z - COLSUM(1; 1, z)+1$ to

$$z - COLSUM(1; 1, z) + ROWSUM(z; 1, \beta) \leqq z - COLSUM(1; 1, z) + 2\tilde{\beta}$$

(see Corollary 2.1.2). Thus, nodes of the same chain will receive distinct last $\lfloor \log_2 \beta \rfloor + 1$ bits for their labels. Because the last $\lfloor \log_2 \beta \rfloor + 1$ bits for the nodes of the same chain are unique and nodes of different chains have different first $\lfloor \log_2 \alpha \rfloor$ bits, each node is given a unique $(\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor + 1)$-bit label by the dilation-3 strategy.

Again, because of the cyclic construction of matrix A and because A contains only 1's and 2's, we have the following formal proofs for the dilation-3 strategy.

If vertical grid-neighbors $[x, y]$ and $[x+1, y]$ belong to the same chain, the following lemma shows that their *MARK* values will differ by exactly 1.

LEMMA 2.3.1. *For all* $1 \leqq x < \alpha$ *and* $1 \leqq y \leqq \beta$ *such that* $CHAIN[x, y] = z = CHAIN[x+1, y]$, $MARK[x, y] = MARK[x+1, y]+1$.

Proof. Let $D = MARK[x, y] - MARK[x+1, y]$. As $CHAIN[x, y] = z$, $D = \delta[x, y] - \delta[x+1, y]$ and $\delta[x, y] = 1$. Since there are at most two grid nodes in each column belonging to the same chain, $[x+2, y]$ cannot be in the same chain with $[x+1, y]$. Thus we have $\delta[x+1, y] = 0$ and $D = 1$. □

If vertical grid-neighbors $[x, y]$ and $[x+1, y]$ belong to different chains, their *MARK* values will be the same as shown in the following lemma.

LEMMA 2.3.2. *For all* $1 \leqq x < \alpha$ *and* $1 \leqq y \leqq \beta$ *such that* $CHAIN[x, y] = z \neq CHAIN[x+1, y]$, $MARK[x, y] = MARK[x+1, y]$.

Proof. Since $CHAIN[x, y] = z \neq CHAIN[x+1, y]$, then $\delta[x, y] = 0$ and, by Lemma 2.2.1, $CHAIN[x+1, y] = z+1$. Thus, we have

$$\begin{aligned} D &= MARK[x+1, y] - MARK[x, y] \\ &= 1 - COLSUM(1; 1, z+1) + COLSUM(1; 1, z) \\ &\quad + ROWSUM(z+1; 1, y-1) - ROWSUM(z; 1, y-1) + \delta[x+1, y]. \end{aligned}$$

Furthermore, we have

$$COLSUM(1; 1, z+1) = COLSUM(1; 1, z) + a_{z+1,1}$$

and, by Lemma 2.1.5, we have

$$ROWSUM(z+1; 1, y-1) = a_{z+1,1} + ROWSUM(z; 1, y-1) - a_{z, y-1}.$$

As $CHAIN[x+1, y] \neq z$, depending on the value of $a_{z+1, y}$, $[x+1, y]$ and $[x+2, y]$ may or may not be in the same chain, i.e., $\delta[x+1, y] = a_{z+1, y} - 1 = a_{z, y-1} - 1$. Thus $D = 0$. □

The following lemma shows that the *MARK* values of horizontal grid-neighbors $[x, y]$ and $[x, y+1]$ will never be the same but that they differ by no more than 2.

LEMMA 2.3.3. *For all* $1 \leqq x \leqq \alpha$ and $1 \leqq y < \beta$,

$$1 \leqq MARK[x, y+1] - MARK[x, y] \leqq 2.$$

Proof. Suppose $CHAIN[x, y] = z$, and let $D = MARK[x, y+1] - MARK[x, y]$.

Case i. $CHAIN[x, y+1] = z$.

$$D = ROWSUM(z; 1, y) - ROWSUM(z; 1, y-1) + \delta[x, y+1] - \delta[x, y]$$
$$= a_{z,y} + \delta[x, y+1] - \delta[x, y].$$

If $\delta[x, y] = 1$, $[x, y]$ and $[x+1, y]$ should belong to the same chain z; in other words, there are 2 nodes in column y of grid G in chain z. Thus, we have $a_{z,y} = 2$ and $1 \leqq D \leqq 2$.

If $\delta[x, y] = 0$ and $a_{z,y} = 1$, then $1 \leqq D \leqq 2$.

If $\delta[x, y] = 0$ and $a_{z,y} = 2$, then $\delta[x, y+1] = 0$. Assume to the contrary that $\delta[x, y+1] = 1$. Since $\delta[x, y] = 0$ and $a_{z,y} = 2$, the topmost node in column y belonging to chain z is $[x-1, y]$; similarly, as $\delta[x, y+1] = 1$ and $CHAIN[x, y+1] = z$, the bottommost node in column $y+1$ belonging to chain z is $[x+1, y+1]$. This contradicts Lemma 2.1.6. Therefore, we have $\delta[x, y+1] = 0$ and $D = 2$.

Case ii. $CHAIN[x, y+1] = z+1$ (Fig. 2.3(a)).

$$D = 1 - COLSUM(1; 1, z+1) + COLSUM(1; 1, z) + ROWSUM(z+1; 1, y)$$
$$- ROWSUM(z; 1, y-1) + \delta[x, y+1] - \delta[x, y].$$

We have $COLSUM(1; 1, z+1) = COLSUM(1; 1, z) + a_{1,z+1}$ and $ROWSUM(z+1; 1, y) = ROWSUM(z; 1, y-1) + a_{1,z+1}$ (by Lemma 2.1.5). Thus,

$$D = 1 + \delta[x, y+1] - \delta[x, y].$$

Note that $\delta[x, y] = 0$. Assume to the contrary that $\delta[x, y] = 1$. Since $\delta[x, y] = 1$, the topmost node in column y belonging to chain z is $[x, y]$. Since $CHAIN[x, y+1] = z+1$, the bottommost node in column $y+1$ belonging to chain z is either $[x-1, y+1]$ or $[x-2, y+1]$. In either case, this contradicts Lemma 2.1.6.

Also, $\delta[x, y+1] = 1$. Assume to the contrary that $\delta[x, y+1] = 0$. Then, the topmost node in column y belonging to chain $z+1$ is $[x+1, y]$. Since $\delta[x, y+1] = 0$, the bottommost node in column $y+1$ belonging to chain $z+1$ is $[x, y+1]$. This contradicts Lemma 2.1.6.

With $\delta[x, y] = 0$ and $\delta[x, y+1] = 1$, $D = 2$.

Case iii. $CHAIN[x, y+1] = z-1$ (Fig. 2.3(b)).

As $COLSUM(1; 1, z) = COLSUM(1; 1, z-1) + a_{z,1}$, and by Lemma 2.1.5,

$$ROWSUM(z-1; 1, y) = ROWSUM(z; 1, y) - a_{z,1} + a_{z-1,y}$$
$$= ROWSUM(z; 1, y-1) + a_{z,y} - a_{z,1} + a_{z,y+1} \text{ (as } a_{z-1,y} = a_{z,y+1}),$$

we have $D = a_{z,y} + a_{z,y+1} - 1 + \delta[x, y+1] - \delta[x, y]$.

Note that $\delta[x, y+1] = 0$. Suppose to the contrary that $\delta[x, y+1] = 1$. Since $CHAIN[x, y] = z$, the topmost node in column y belonging to chain z is either $[x, y]$ or $[x-1, y]$. Since $\delta[x, y+1] = 1$ and $CHAIN[x, y+1] = z-1$, the bottommost node in column $y+1$ belonging to chain z is either $[x+2, y+1]$ or $[x+3, y+1]$. This contradicts Lemma 2.1.6.

Also, note that $a_{z,y+1} = 1$. Suppose to the contrary that $a_{z,y+1} = 2$. Since $CHAIN[x, y] = z$, the topmost node in column y belonging to chain z is either $[x, y]$ or $[x-1, y]$. Since $CHAIN[x, y+1] = z-1$ and $a_{z,y+1} = 2$, the bottommost node in column $y+1$ belonging to chain z is $[x+2, y+1]$. This contradicts Lemma 2.1.6.

If $\delta[x, y] = 0$, then $a_{z,y} = 1$. Suppose to the contrary that $a_{z,y} = 2$. Since $\delta[x, y] = 0$ and $a_{z,y} = 2$, the topmost node in column y belonging to chain z is $[x-1, y]$. If $\delta[x, y+1] = 0$ and $CHAIN[x, y+1] = z-1$, the bottommost node in column $y+1$ belonging to chain z is either $[x+1, y+1]$ or $[x+2, y+1]$. This contradicts Lemma 2.1.6. Thus, $a_{z,y} = 1$. With $\delta[x, y+1] = 0$ and $a_{z,y+1} = 1$, $D = 1$.

If $\delta[x, y]=1$, then $a_{z,y}=2$. With $\delta[x, y+1]=0$ and $a_{z,y+1}=1$, $D=1$. □

COROLLARY 2.3.1. *The last* $\lfloor \log_2 \beta \rfloor+1$ *bits of the labels assigned to grid-neighbors will differ in at most two bit positions.*

Proof. Lemmas 2.3.1, 2.3.2, and 2.3.3 show that the *MARK* values of grid-neighbors differ by at most two. Since the last $\lfloor \log_2 \beta \rfloor+1$ bits of the label assigned to $[x, y]$ is $GRAY(\lfloor \log_2 \beta \rfloor+1, MARK[x, y])$, the last $\lfloor \log_2 \beta \rfloor+1$ bits of grid-neighbors will differ in at most two bit positions. □

Corollaries 2.2.1 and 2.3.1 combined tell us that the dilation-3 embedding strategy does indeed yield a dilation of at most 3. To achieve a dilation-2 embedding, our strategy is to modify the last bit. In view of this, we make the following observations about the first $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ bits of the $(\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor+1)$-bit label.

LEMMA 2.4.1. *The first* $\lfloor \log_2 \beta \rfloor$ *bits of the last* $\lfloor \log_2 \beta \rfloor+1$ *bits of the labels assigned to grid-neighbors will differ in at most one bit position.*

Proof. Consider any pair of grid-neighbors. If they differ in at most one bit position in the last $\lfloor \log_2 \beta \rfloor+1$ bits of their labels, then clearly the first $\lfloor \log_2 \beta \rfloor$ bits of the last $\lfloor \log_2 \beta \rfloor+1$ bits of their labels will differ in at most one bit position. Otherwise, by Corollary 2.3.1, it must be the case that the grid-neighbors differ in exactly two bit positions of the last $\lfloor \log_2 \beta \rfloor+1$ bits of their labels, and moreover, their *MARK* values differ by exactly two. Since binary-reflected Gray code is used, the grid-neighbors will have different last bits; thus, the first $\lfloor \log_2 \beta \rfloor$ bits of the last $\lfloor \log_2 \beta \rfloor+1$ bits of the labels assigned to grid-neighbors will differ in at most one bit position. □

COROLLARY 2.4.1. *The first* $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ *bits of the labels assigned to grid-neighbors will differ in at most two bit positions.*

Proof. By Corollary 2.2.1, the first $\lfloor \log_2 \alpha \rfloor$ bits of the labels assigned to grid-neighbors will differ in at most one bit position. With Lemma 2.4.1, the corollary follows. □

COROLLARY 2.4.2. *If the first* $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ *bits of the labels assigned to grid-neighbors differ in exactly two bit positions, the two grid-neighbors must be assigned to different chains.*

Proof. Since the first $\lfloor \log_2 \alpha \rfloor$ bits of the labels must differ in one bit position, the two grid-neighbors must be assigned to adjacent chains. □

LEMMA 2.4.2. *At most two nodes will have the same first* $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ *bits for their labels.*

Proof. Since each node is given a unique $(\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor+1)$-bit label by the dilation-3 strategy, at most two nodes will have the same first $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ bits for their labels. □

DILATION-2 EMBEDDING STRATEGY

(1) Assume, without loss of generality, that $\alpha \leqq 3\tilde{\alpha}/2$. (Either $\alpha \leqq 3\tilde{\alpha}/2$ or $\beta \leqq 3\tilde{\beta}/2$; for otherwise, $\alpha\beta > 9\tilde{\alpha}\tilde{\beta}/4 > 2\tilde{\alpha}\tilde{\beta}$.) Construct $A(\alpha, \beta)$.

(2)–(5) as in the dilation-3 strategy.

(6) Modify the last bit of each node so that

(a) two nodes with the same first $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ bits will differ in their last bit, and

(b) two grid-neighbors with first $\lfloor \log_2 \alpha \rfloor+\lfloor \log_2 \beta \rfloor$ bits which differ in exactly two bit positions will have the same last bit.

By design, this strategy forces a dilation of at most two via step (6). What remains is to show that step (6) can always be accomplished. To this end, we will construct a dependency graph $G'=(V, E)$ which has as its nodes the nodes of G, i.e.,

$$V(G')=\{[x, y] \mid 1 \leqq x \leqq \alpha, 1 \leqq y \leqq \beta\}.$$

There is an edge between two nodes in G' if and only if either they have the same first $\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor$ bits, or they are grid-neighbors whose first $\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor$ bits differ in exactly two bit positions. An edge between nodes $[x, y]$ and $[u, v]$ indicates that the last bit assigned to $[x, y]$ will affect the last bit assigned to $[u, v]$, and vice versa, based on the idea that distinct labels are assigned to different nodes and grid-neighbors are within dilation 2 of each other. Certainly, if G' is acyclic, then step (6) can be accomplished. The assumption made in step (1) will help in proving G' to be acyclic. Note that some of the claims in the remainder of this section only hold with this assumption. Figure 2.4 shows G' for the 11×11 grid.

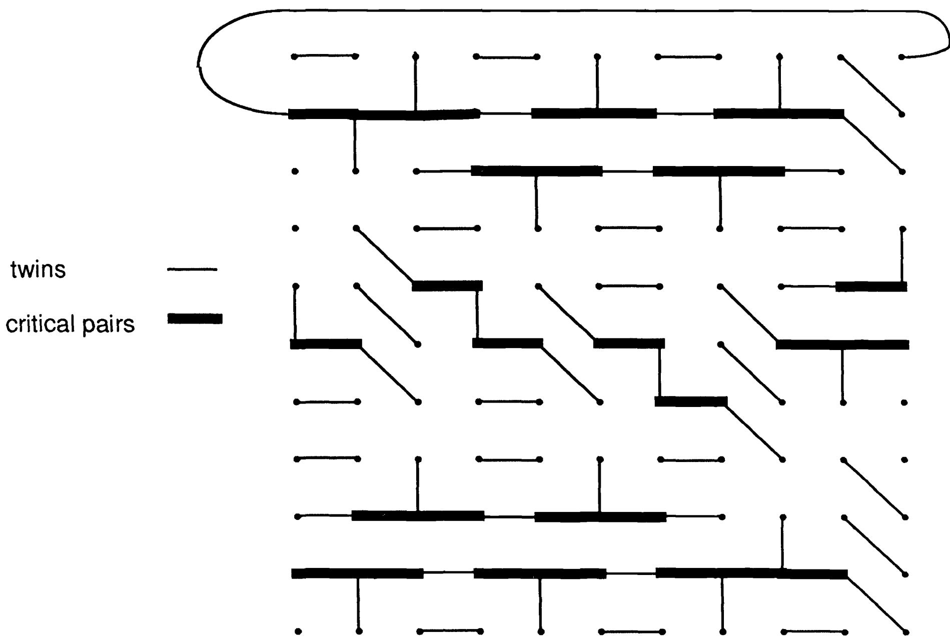


FIG. 2.4. *Dependency graph G' for an 11×11 grid.*

We introduce some additional terminology. Nodes $[x, y]$ and $[u, v]$ are said to be *twins* if and only if they have the same first $\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor$ bits. Nodes $[x, y]$ and $[u, v]$ are said to be *a critical pair* if and only if they are grid-neighbors whose first $\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor$ bits differ in exactly two bit positions. Thus, the edges of G', $E(G')$, are either between twins or critical pairs. By the definition of twins and Corollary 2.4.2, twins must belong to the same chain while critical pairs are in different chains. A distinction is made between edges joining twins and edges joining critical pairs in Fig. 2.4. Note that edges joining critical pairs are all between horizontal grid-neighbors, i.e., between $[x, y]$ and $[x, y+1]$, belonging to different chains. This observation is captured by the following lemma.

LEMMA 2.5.1. *If $[x, y]$ and $[u, v]$ are a critical pair, then $[u, v] \in \{[x, y-1], [x, y+1]\}$.*

Proof. Because $[x, y]$ and $[u, v]$ are a critical pair, by definition they must be grid-neighbors, i.e.,

$$[u, v] \in \{[x, y-1], [x, y+1], [x-1, y], [x+1, y]\},$$

and by Corollary 2.4.2 they must belong to different chains. Assume that $[x, y]$ and $[u, v]$ are vertical grid-neighbors. By Lemma 2.3.2, their *MARK* values must be the same, making it impossible for them to be a critical pair. Therefore, $[u, v]$ and $[x, y]$ must be horizontal grid-neighbors. □

Note also that edges joining twins are line segments of chains, with the exception of "wraparound" edges. An edge between twins $[x, y]$ and $[u, v]$ in G' is said to be a *wraparound edge* if and only if $y = 1$ and $v = \beta$, i.e., a wraparound edge connects a node in the first column of G' with a node in the last. Thus, upon inspecting the edges shown in Fig. 2.4, we find that, apart from wraparound edges, the edges in G' are either between horizontal grid-neighbors (i.e., $[x, y]$ and $[x, y+1]$) or vertical grid-neighbors (i.e., $[x, y]$ and $[x+1, y]$), or that they slope down from $[x, y]$ to $[x+1, y+1]$. This observation is captured by the following lemma.

LEMMA 2.5.2. *If edge* $\{[x, y], [u, v]\} \in E(G')$ *is not a wraparound edge, then*

$$[u, v] \in \{[x, y-1], [x, y+1], [x-1, y-1], [x-1, y], [x+1, y], [x+1, y+1]\}.$$

Proof. If edge $\{[x, y], [u, v]\} \in E(G')$, then either $[x, y]$ and $[u, v]$ are twins, or $[x, y]$ and $[u, v]$ are critical pairs. If $[x, y]$ and $[u, v]$ are critical pairs, then, by Lemma 2.5.1, $[u, v] \in \{[x, y-1], [x, y+1]\}$. What remains is to consider the case when $[x, y]$ and $[u, v]$ are twins and the edge between them is not a wraparound edge. Thus, $CHAIN[x, y] = CHAIN[u, v]$, and $MARK[x, y]$ and $MARK[u, v]$ differ by 1. From the way in which nodes are given *MARK* values, $[x, y]$ and $[u, v]$ must be joined by a line segment of $CHAIN[x, y]$. Thus $[u, v]$ can neither be $[x-1, y+1]$ nor $[x+1, y-1]$. □

There are various useful properties associated with a wraparound edge joining twins $[x, 1]$ and $[u, \beta]$:

(i) $ROWSUM(CHAIN[x, 1]; 1, \beta) = 2\tilde{\beta}$ because $[x, 1]$ and $[u, \beta]$ must be the two end-nodes of the same chain and must be exactly $2\tilde{\beta} - 1$ apart in order to have the first $\lfloor \log_2 \alpha \rfloor + \lfloor \log_2 \beta \rfloor$ bits of their labels be the same.

(ii) If $1 \leqq x < \tilde{\alpha}$, $CHAIN[x+1, 1] \neq CHAIN[x, 1] = CHAIN[u, \beta]$ by the same argument as in (i), thus we must have $\delta[x, 1] = 0$ in order that $NUMBER[x, 1] = 1$.

(iii) If $1 < u \leqq \tilde{\alpha}$, $CHAIN[x, 1] = CHAIN[u, \beta] \neq CHAIN[u-1, \beta]$ by the same argument as in (i); thus we must have $\delta[u-1, \beta] = 0$ in order that $NUMBER[u, \beta]$ be the largest among all nodes in the chain.

In Fig. 2.4, there is only one wraparound edge: $\{[2, 1], [1, 11]\}$. The reader can easily check, by looking back at Fig. 2.1(b), that the chain for $[2, 1]$ and $[1, 11]$, i.e. chain 1, does indeed have 16 nodes in it and node $[3, 1]$ does not belong to chain 1.

The proof that G' is acyclic proceeds by contradiction. Suppose G' has a simple cycle. Let the sequence of nodes in the cycle $[u_1, v_1]$, $[u_2, v_2], \cdots, [u_{m-1}, v_{m-1}]$, $[u_m, v_m]$, $[u_1, v_1]$, where $[u_1, v_1]$ is such that, for all $2 \leqq i \leqq m$, $u_1 \leqq u_i$ and if $u_1 = u_i$, then $v_1 < v_i$. In other words, we let the leftmost (least column-wise) from among the topmost (least row-wise) nodes in the cycle be deemed the first node $[u_1, v_1]$ of the cycle. We shall prove that, if $[u_1, v_1]$ is the first node of the cycle, then, without loss of generality, $\{[u_1, v_1], [u_m, v_m]\}$ must be a wraparound edge. Furthermore, if $\{[u_1, v_1], [u_m, v_m]\}$ is a wraparound edge and $[u_1, v_1]$ is the leftmost among the topmost nodes in the cycle, then $a_{CHAIN[u_1, v_1], \beta} = 1$, which implies that the number of nodes in $CHAIN[u_1, v_1]$ is less than $2\tilde{\beta}$. Thus, this leads to a contradiction of the fact that a chain with a wraparound edge is of maximum length $2\tilde{\beta}$.

Note that either $[u_1, v_1]$ and $[u_2, v_2]$ are a critical pair, or $[u_1, v_1]$ and $[u_m, v_m]$ are a critical pair, since both pairs cannot be twins. So, we can assume without loss of generality, that $[u_1, v_1]$ and $[u_2, v_2]$ are a critical pair (otherwise, consider $[u_1, v_1]$,

$[u_m, v_m]$, $[u_{m-1}, v_{m-1}], \cdots, [u_2, v_2]$, $[u_1, v_1]$ instead), implying that $[u_2, v_2] \in \{[u_1, v_1-1], [u_1, v_1+1]\}$ (by Lemma 2.5.1). However, since $[u_1, v_1]$ is leftmost from among the topmost nodes in the cycle, then $[u_2, v_2]=[u_1, v_1+1]$. Note that since $v_1 < v_2$, $[u_1, v_1]$ cannot be in the last column, i.e., $v_1 < \beta$.

Next, we wish to argue that $\{[u_1, v_1], [u_m, v_m]\} \in E(G')$ must be a wraparound edge. In so doing, we make use of the following convenient fact.

LEMMA 2.5.3. *When* $\alpha \leqq 3\tilde{\alpha}/2$, *for all* $1 \leqq j < \beta$, $a_{1,j} + a_{1,j+1} \leqq 3$.

Proof.

$$a_{1,j} + a_{1,j+1} = \begin{cases} \lceil \alpha/\tilde{\alpha} \rceil + \lfloor (\tilde{\alpha}-1)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-2)\alpha/\tilde{\alpha} \rfloor & \text{if } j=1 \\ \lfloor (\tilde{\alpha}-j+1)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha}-j-1)\alpha/\tilde{\alpha} \rfloor & \text{if } 1<j<\tilde{\alpha} \\ \lfloor \alpha/\tilde{\alpha} \rfloor + \lceil \alpha/\tilde{\alpha} \rceil & \text{if } j=\tilde{\alpha} \\ a_{1,j-\tilde{\alpha}} + a_{1,j+1-\tilde{\alpha}} & \text{if } j>\tilde{\alpha}. \end{cases}$$

Using (1) and (2), we can show that $a_{1,j} + a_{1,j+1} \leqq \lceil 2\alpha/\tilde{\alpha} \rceil$. The lemma follows since $\alpha \leqq 3\tilde{\alpha}/2$. □

LEMMA 2.5.4. $\{[u_1, v_1], [u_m, v_m]\} \in E(G')$ *must be a wraparound edge.*

Proof. Assume to the contrary that $\{[u_1, v_1], [u_m, v_m]\} \in E(G')$ is not a wraparound edge. Then, by Lemma 2.5.2,

$$[u_m, v_m] \in \{[u_1, v_1-1], [u_1, v_1+1],$$
$$[u_1-1, v_1-1], [u_1-1, v_1], [u_1+1, v_1], [u_1+1, v_1+1]\}.$$

However, since $[u_1, v_1]$ is leftmost from among the topmost nodes in the cycle and $[u_m, v_m] \neq [u_1, v_1+1] = [u_2, v_2]$, then $[u_m, v_m] \in \{[u_1+1, v_1], [u_1+1, v_1+1]\}$. By Lemma 2.5.1, we know that $[u_1, v_1]$ and $[u_m, v_m]$ are twins. So, $[u_{m-1}, v_{m-1}]$ and $[u_m, v_m]$ are a critical pair and by Lemma 2.5.1, $[u_{m-1}, v_{m-1}] \in \{[u_m, v_m-1], [u_m, v_m+1]\}$. The remaining possibilities can now separate into three cases, all of which will be shown to be impossible.

Case a. $[u_m, v_m]=[u_1+1, v_1+1]$. (See Fig. 2.5(a).)

Because $[u_1, v_1]$ and $[u_m, v_m]$ are twins, the first $\lfloor \log_2 \beta \rfloor$ bits of the last $\lfloor \log_2 \beta \rfloor + 1$ bits of the labels of $[u_1, v_1]$ and $[u_m, v_m]$ must be equal. Since $[u_m, v_m]=[u_2+1, v_2]$, by Lemma 2.3.1, $MARK[u_m, v_m]= MARK[u_2, v_2]$ and the first $\lfloor \log_2 \beta \rfloor$ bits of the last $\lfloor \log_2 \beta \rfloor + 1$ bits of the labels of $[u_m, v_m]$ and $[u_2, v_2]$ must also be equal. This makes it impossible for $[u_1, v_1]$ and $[u_2, v_2]$ to be a critical pair. Thus, this case is impossible.

Case b. $[u_m, v_m]=[u_1+1, v_1]$ and $[u_{m-1}, v_{m-1}]=[u_m, v_m+1]$. (See Fig. 2.5(b).)

Since $[u_2, v_2]$ and $[u_1, v_1]$ are a critical pair, $[u_1, v_1]$ and $[u_m, v_m]$ are twins, and $[u_m, v_m]$ and $[u_{m-1}, v_{m-1}]$ are a critical pair, $CHAIN[u_2, v_2]$ and $CHAIN[u_1, v_1]$ differ by one, $CHAIN[u_1, v_1]= CHAIN[u_m, v_m]$, and $CHAIN[u_m, v_m]$ and $CHAIN[u_{m-1}, v_{m-1}]$ differ by one. However, $[u_2, v_2]$ and $[u_{m-1}, v_{m-1}]$ are grid-neighbors, and hence, by Lemma 2.2.1, $CHAIN[u_2, v_2]$ and $CHAIN[u_{m-1}, v_{m-1}]$ should

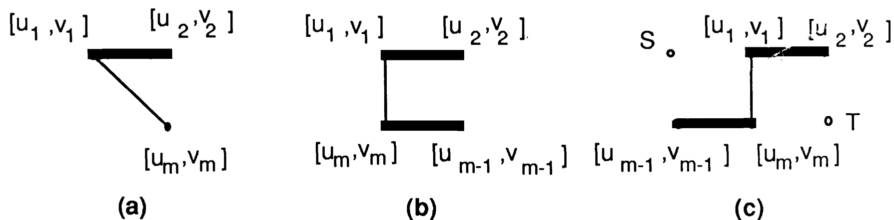


FIG. 2.5. *Impossible cases in G′ for cycle formation.*

differ by at most one. This means that $CHAIN[u_2, v_2] = CHAIN[u_{m-1}, v_{m-1}]$. Hence, the edge joining the topmost node of column v_1 belonging to $CHAIN[u_1, v_1]$ and the bottommost node of column $v_2 = v_1 + 1$ belonging to the same chain must slope upward or slope downward too much, violating Lemma 2.1.6. Thus, this case is also impossible.

Case c. $[u_m, v_m] = [u_1 + 1, v_1]$ and $[u_{m-1}, v_{m-1}] = [u_m, v_m - 1]$. (See Fig. 2.5(c).)

First note that nodes S and T in Fig. 2.5(c) must belong to the same chain as $[u_1, v_1]$ and $[u_m, v_m]$ (for otherwise, Lemma 2.1.6 would be violated, i.e., the line segments of $CHAIN[u_1, v_1]$ from column $v_1 - 1$ to v_1 and from column v_1 to $v_1 + 1$ will slope upward or slope downward too much). This means that

$$COLSUM(v_1 - 1; 1, CHAIN[u_1, v_1]) = u_1, \quad COLSUM(v_1; 1, CHAIN[u_1, v_1]) = u_1 + 1$$

and

$$COLSUM(v_1 + 1; 1, CHAIN[u_1, v_1]) \geqq u_1 + 1$$

(because $CHAIN[u_{m-1}, v_{m-1}] \neq CHAIN[S]$), which implies that $a_{1,v_1} = 2$ and $a_{1,v_1+1} = 2$ (by Corollaries 2.1.4 and 2.1.3). This contradicts Lemma 2.5.3. □

Given that $\{[u_1 v_1], [u_m, v_m]\}$ is a wraparound edge and because $v_1 < \beta$, we have $v_1 = 1$ and $v_m = \beta$. Thus, from now on, we denote $\{[u_1, 1], [u_m, \beta]\}$ as the wraparound edge. The next five lemmas concern *ROWSUM*s and *COLSUM*s within matrix A, with particular attention paid to row 1, row $CHAIN[u_1, 1]$, and columns 1, $v_2 \equiv 2$, $v_{m-1} \equiv \beta - 1$ and β of A. Taking full advantage of $\{[u_1, 1], [u_m, \beta]\}$ being a wraparound edge, with these lemmas we finally arrive at the contradiction that G' contains a cycle.

The following lemma shows that the number of nodes in columns 1 to k of the first chain is at least as many as the number of nodes in columns 1 to k of $CHAIN[u_1, 1]$ for any $1 \leqq k \leqq \beta$.

LEMMA 2.6.1. *For all* $1 \leqq k \leqq \beta$,

$$ROWSUM(CHAIN[u_1, 1]; 1, k) \leqq ROWSUM(1; 1, k) = \lceil k\alpha/\tilde{\alpha} \rceil.$$

Proof. Let $k = p\tilde{\alpha} + q$ where p, q are integers and $0 \leqq q < \tilde{\alpha}$.

$$\begin{aligned} ROWSUM(1; 1, k) &= p\alpha + ROWSUM(1; 1, q) \\ &= p\alpha + \lceil \alpha/\tilde{\alpha} \rceil + \lfloor (\tilde{\alpha} - 1)\alpha/\tilde{\alpha} \rfloor - \lfloor (\tilde{\alpha} - q)\alpha/\tilde{\alpha} \rfloor \\ &= p\alpha + \alpha - \lfloor (\tilde{\alpha} - q)\alpha/\tilde{\alpha} \rfloor = p\alpha + \lceil q\alpha/\tilde{\alpha} \rceil = \lceil k\alpha/\tilde{\alpha} \rceil, \end{aligned}$$

and from Lemma 2.1.3, $ROWSUM(CHAIN[u_1, 1]; 1, k) \leqq \lceil k\alpha/\tilde{\alpha} \rceil$. □

The next lemma shows that the chain with the wraparound edge and the first chain are both of maximum length.

LEMMA 2.6.2. $ROWSUM(CHAIN[u_1, 1]; 1, \beta) = ROWSUM(1; 1, \beta) = 2\tilde{\beta}$.

Proof. Because $\{[u_1, 1], [u_m, \beta]\}$ is a wraparound edge, $ROWSUM(CHAIN[u_1, 1]; 1, \beta) = 2\tilde{\beta}$. From Lemma 2.6.1, we have

$$ROWSUM(CHAIN[u_1, 1]; 1, \beta) \leqq ROWSUM(1; 1, \beta) \leqq \lceil \beta\alpha/\tilde{\alpha} \rceil \leqq 2\tilde{\beta}.$$

The lemma follows. □

LEMMA 2.6.3. $COLSUM(1; 1, CHAIN[u_1, 1]) = u_1$ *and* $COLSUM(2; 1, CHAIN[u_1, 1]) = u_1 - 1$.

Proof. Because $\{[u_1, 1], [u_m, \beta]\}$ is a wraparound edge, $CHAIN[u_1 + 1, 1] \neq CHAIN[u_1, 1]$, and thus, $COLSUM(1; 1, CHAIN[u_1, 1]) = u_1$. Then, from Lemma 2.1.2,

$$COLSUM(2; 1, CHAIN[u_1, 1]) \in \{u_1 - 1, u_1, u_1 + 1\}.$$

Since $[u_1, 1]$ and $[u_2, 2]$ are a critical pair, $CHAIN[u_2, 2] \neq CHAIN[u_1, 1]$ and thus,

$$COLSUM(2; 1, CHAIN[u_1, 1]) \neq u_1 = u_2.$$

Suppose $COLSUM(2; 1, CHAIN[u_1, 1]) = u_1 + 1$. $COLSUM(1; 1, CHAIN[u_1, 1]) = u_1$ and $COLSUM(2; 1, CHAIN[u_1, 1]) = u_1 + 1$ imply that $a_{1,2} = 2$ (Corollary 2.1.4). Since $a_{1,1} = \lceil \alpha / \tilde{\alpha} \rceil = 2$, $a_{1,1} a_{1,2} = 4$ contradicts Lemma 2.5.3. Thus, $COLSUM(2; 1, CHAIN[u_1, 1]) = u_1 - 1$. □

LEMMA 2.6.4.

$$COLSUM(\beta; 1, CHAIN[u_1, 1]) = u_1 = u_m$$

and

$$COLSUM(\beta - 1; 1, CHAIN[u_1, 1]) = u_1 - 1.$$

Proof. By Lemmas 2.6.3 and 2.1.2,

$$COLSUM(\beta - 1; 1, CHAIN[u_1, 1]), COLSUM(\beta; 1, CHAIN[u_1, 1]) \in \{u_1 - 1, u_1\}.$$

Because $\{[u_1, 1], [u_m, \beta]\}$ is a wraparound edge, $CHAIN[u_m, \beta] = CHAIN[u_1, 1]$. So, $COLSUM(\beta; 1, CHAIN[u_1, 1]) \geqq u_m$. $u_m \geqq u_1$ since $[u_1, 1]$ is topmost. Thus,

$$COLSUM(\beta; 1, CHAIN[u_1, 1]) = u_m = u_1.$$

Since $CHAIN[u_{m-1}, \beta - 1] \neq CHAIN[u_m, \beta]$, then $COLSUM(\beta - 1; 1, CHAIN[u_1, 1]) \neq u_1$. Thus,

$$COLSUM(\beta - 1; 1, CHAIN[u_1, 1]) = u_1 - 1.$$ □

LEMMA 2.6.5. $a_{1,\beta} = 2$ *and* $a_{CHAIN[u_1,1],\beta} = 1$.

Proof. Lemma 2.6.4 implies that $a_{1,\beta} = 2$ by Corollary 2.1.4. Because $\{[u_1, 1], [u_m, \beta]\}$ is a wraparound edge,

$$CHAIN[u_m - 1, \beta] \neq CHAIN[u_m, \beta] = CHAIN[u_1, 1].$$

Lemma 2.6.4 says that

$$COLSUM(\beta; 1, CHAIN[u_1, 1]) = u_1 = u_m.$$

Thus, $a_{CHAIN[u_1,1],\beta} = a_{CHAIN[u_m,\beta],\beta} = 1$. □

There is the following contradiction: Lemmas 2.6.1 and 2.6.5 imply that

$$\begin{aligned} &ROWSUM(CHAIN[u_1, 1]; 1, \beta) \\ &\quad = ROWSUM(CHAIN[u_1, 1]; 1, \beta - 1) + a_{CHAIN[u_1,1],\beta} \\ &\quad \leqq ROWSUM(1; 1, \beta - 1) + a_{CHAIN[u_1,1],\beta} \\ &\quad \leqq ROWSUM(1; 1, \beta) - a_{1,\beta} + a_{CHAIN[u_1,1],\beta} \leqq 2\tilde{\beta} - 1, \end{aligned}$$

which contradicts Lemma 2.6.2. Thus G' is acyclic.

Since G' is acyclic, step (6) of the algorithm can always be done. Implementation-wise, step (6) can be done by first building dependency graph G' and then traversing G' by either using a breadth-first or depth-first tree traversal scheme. When each node T is visited for the first time, its last bit is determined. Suppose T's parent in the breadth-first or depth-first tree is S. If S and T are twins, then make T's last bit different from S's. If, on the other hand, S and T are a critical pair, then make T's last bit the same as S's.

Let us now look at the overall time complexity for our dilation-2 embedding strategy. Computing the $\tilde{\alpha} \times \beta$ matrix $A(\alpha, \beta)$ takes $O(\tilde{\alpha}\beta)$ time since each element of A can be computed in constant time. Determining the first $\lfloor \log_2 \alpha \rfloor$ bits of the labels for all nodes $[x, y]$, $1 \leqq x \leqq \alpha$ and $1 \leqq y \leqq \beta$, takes $O(\alpha\beta)$ time since $CHAIN[x, y]$ for all nodes $[x, y]$ can be computed in $O(\alpha\beta)$ time. Determining the last $\lfloor \log_2 \beta \rfloor + 1$ bits of the labels for all nodes $[x, y]$ also takes $O(\alpha\beta)$ time since $MARK[x, y]$ for all nodes $[x, y]$ can be computed in $O(\alpha\beta)$ time, and modifying the last bit for all nodes $[x, y]$ takes $O(\alpha\beta)$ time since the construction and traversal of G', which has $\alpha\beta$ nodes and $O(\alpha\beta)$ edges, takes $O(\alpha\beta)$ time. Thus, we have the following theorem.

THEOREM 2.1. *Any two-dimensional $\alpha \times \beta$ grid can be embedded into its optimal hypercube with a dilation of at most 2 in $O(\alpha\beta)$ time.*

3. The d-dimensional embedding strategy. Having obtained the best possible dilation for all two-dimensional grids, the next question is: how can the technique be extended to higher-dimension grids? Trivial extensions to higher dimensions, unfortunately, tend to cause grids to be mapped to larger-than-optimal-sized hypercubes. The insistence on embedding into optimal hypercubes makes the problem nontrivial and requires a careful embedding strategy. In this section, we give the following result: all d-dimensional grids can be embedded in their optimal hypercubes with dilation at most $4d+1$.

The optimal hypercube for an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional grid is a $\lceil \log_2 n_1 n_2 \cdots n_d \rceil$-cube. So, given an $n_1 \times n_2 \times \cdots \times n_d$ d-dimensional grid X, we want to ultimately map each node of X to a $\lceil \log_2 n_1 n_2 \cdots n_d \rceil$-bit label. While our two-dimensional strategy consists of two stages, our d-dimensional embedding strategy consists of d stages: at each stage some bits of each node's label are determined. Once again, partitioning the nodes plays a key role. Initially, all nodes of X form a single group. At each successive stage, each group of nodes is in turn partitioned into a power-of-two number of groups of about equal size. If, at some stage, each group is partitioned into say 2^m groups, then to which of the 2^m groups a particular node belongs determines m additional bits of its label. After d stages, each node will have a unique label. The partitioning is done in such a way as to ensure that grid-neighbors belonging to different groups will be given additional bits that differ by no more than four or five bits; thus, after p stages, neighboring nodes in X have accumulated labels that differ by at most $O(p)$ bits. The partitioning process is described in terms of *chains* and *jagged grids*, which are introduced in the next section. Jagged grids relate to our "group" concept. "Chains" partition our groups into smaller groups with about the same number of nodes. Initially, all nodes in the d-dimensional grid are mapped to a jagged grid, a two-dimensional grid-like structure. Thus, the partitioning is performed on a two-dimensional structure rather than a d-dimensional structure.

3.1. Preliminaries: Chains and jagged grids. Essentially, jagged grids are two-dimensional layouts or arrangements of the nodes of X. Jagged grids play a very important part in our embedding strategy and have very interesting properties. At each of the d stages of our embedding algorithm, the jagged grids we deal with will be "compatible," i.e., they will be similar in size and shape and will have similar properties. The process used to partition jagged grids is like that of two-dimensional grids. Again, a partitioning matrix is used to define "chains" to partition the nodes of the jagged grid in such a way that grid-neighbors are assigned to nearby chains and the number of nodes in each of the chains differ by no more than 1. After the chains are determined, nodes within a chain are then laid out to form a smaller jagged grid, which can again recursively be partitioned into smaller chains and jagged grids. The newly-constructed

jagged grids should be compatible and will try to bring nodes in a chain which are grid-neighbors under a particular dimension closer together for further partitioning.

DEFINITION. An (α, β)*-jagged grid* G is a collection of β columns of nodes, each column having at least $\alpha \geqq 2$ and at most $\alpha+3$ nodes; moreover, for any two k-column slices S and S' of G, $|size(S)-size(S')| \leqq 3$, where $1 \leqq k \leqq \beta$. A *k-column slice S* of a jagged grid is defined to be a subcollection of k consecutive columns and $size(S)$ is defined to be the number of nodes in the slice. □

DEFINITION. Let $size(G)$ denote the number of nodes in the jagged grid G. Two (α, β)-jagged grids G and G' are said to be *compatible* if $|size(G)-size(G')| \leqq 1$ and for any k-column slice S taken from G and any k-column slice S' taken from G', $|size(S)-size(S')| \leqq 3$, where $1 \leqq k \leqq \beta$. □

Figure 3.1(a) gives an example of a (5, 6)-jagged grid. There are six columns, each column with at least five and at most eight nodes. In this jagged grid, the sizes of the 2-column slices spanning columns 1 to 2, 2 to 3, 3 to 4, 4 to 5, and 5 to 6 are 12, 11, 13, 13, and 12, respectively. The sizes of the 3-column slices spanning columns 1 to 3, 2 to 4, 3 to 5, and 4 to 6 are 17, 19, 18, and 20, respectively. The sizes of the 4-column slices spanning columns 1 to 4, 2 to 5, and 3 to 6 are 25, 24, and 25, respectively. The sizes of the 5-column slices spanning columns 1 to 5 and 2 to 6 are 30 and 31, respectively. Thus the sizes of any two 2-column slices will differ by at most 3 (likewise, for any two 1-column, 3-column, 4-column, or 5-column slices). There is only one 6-column slice.

Henceforth, we shall, for convenience, refer to the nodes of a jagged grid by its *column* and *row-position* within the column. So, for example, in Fig. 3.1(a), we refer to node U as the node in row-position 1 of column 1, node V as the node in row-position 2 of column 1, and node W as the node in row-position 5 of column 4.

Figure 3.1(b) gives another example of a (5, 6)-jagged grid which happens to be compatible with the (5 6)-jagged grid shown in Fig. 3.1(a). The reader can verify that the two jagged grids are indeed compatible with each other.

Five lemmas are introduced in this section. Lemmas 3.1.1 and 3.1.2 ensure that a jagged grid can be partitioned into power-of-two smaller and compatible jagged grids. Lemma 3.1.3 ensures that, when two compatible jagged grids are each broken down into smaller jagged grids, the smaller jagged grids are all also compatible with one another. Finally, Lemmas 3.1.4 and 3.1.5 make a statement as to the dilation induced by such a partitioning process.

We first describe the procedure for constructing chains within (α, β)-jagged grids. This allows for the systematic partitioning of the nodes in the jagged grid.

Procedure for construction chains. We construct $\tilde{\alpha}=2^{\lfloor \log_2 \alpha \rfloor} \geqq 2$ chains within an (α, β)-jagged grid G. To do so, we define three $\tilde{\alpha} \times \beta$ matrices A, B, and C, where matrix A is simply the matrix $A(\alpha, \beta)$ introduced in § 2 for drawing chains in a two-dimensional $\alpha \times \beta$ grid, matrix B is defined below, and matrix C is simply $A+B$.

For $k_1 \leqq k_2$, let

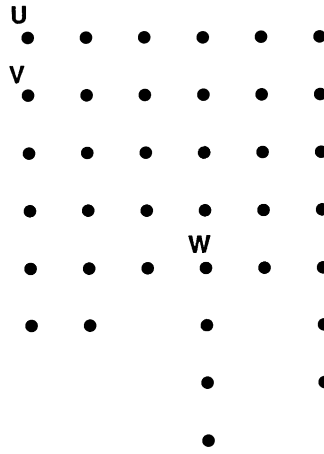
$$ROWSUM(M, i; k_1, k_2) \equiv \sum_{j=k_1}^{k_2} m_{i,j}$$

and

$$COLSUM(M, j; k_1, k_2) \equiv \sum_{i=k_1}^{k_2} m_{i,j}$$

where $m_{i,j}$ denotes the element in the ith row, jth column of matrix M. For $k_1 > k_2$,

$$ROWSUM(M, i; k_1, k_2) \equiv COLSUM(M, j; k_1, k_2) \equiv 0.$$



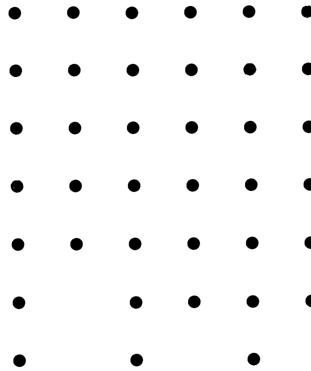


FIG. 3.1. *An example of two compatible* $(5, 6)$*-jagged grids.*

Matrix C will be used in a way similar to how matrix A was used in § 2 to define chains; in particular, $c_{i,j}$ will tell us how many nodes from column j of the jagged grid G will belong to chain i. Thus, matrix C ought to be an $\tilde{\alpha} \times \beta$ matrix since we wish to partition the β columns of G into $\tilde{\alpha}$ chains. To ensure that all of the nodes of each column of G are assigned to chains, we want $COLSUM(C, j; 1, \tilde{\alpha})$ to equal the number of nodes in column j of G. To ensure that the number of nodes in each chain is about the same, we want

$$ROWSUM(C, i; 1, \beta) \in \{\lfloor n/\tilde{\alpha} \rfloor, \lceil n/\tilde{\alpha} \rceil\}$$

where $n = size(G)$.

If each column of jagged grid C had α nodes in it, matrix A could immediately be used to define chains; however, since a column of G may have up to $\alpha + 3$ nodes, matrix A needs to be adjusted before it can adequately define chains in jagged grid

G. Matrix B acts as an adjustment matrix which accounts for the "extra" nodes in the columns of G. With

$$COLSUM(B, j; 1, \tilde{\alpha}) = (\text{the number of nodes in column } j \text{ of } G) - \alpha \leqq 3,$$

then clearly, since $C = A + B$, $COLSUM(C, j; 1, \tilde{\alpha}) =$ (the number of nodes in column j of G). Moreover, we construct matrix B to distribute the "extra" nodes of each column evenly among the $\tilde{\alpha}$ chains, one at a time according to some cyclic order of the chains, starting with the shorter chains first to make sure that

$$ROWSUM(C, i; 1, \beta) \in \{\lfloor n/\tilde{\alpha} \rfloor, \lceil n/\tilde{\alpha} \rceil\}$$

where $n = size(G)$. To construct B, we apply the following procedure:

(i) Initialize B with all 0's.
(ii) Let $r_0, r_1, \cdots, r_{\tilde{\alpha}-1}$ be the row numbers $1, 2, \cdots, \tilde{\alpha}$, ordered in such a way that $ROWSUM(A, r_i; 1, \beta) \leqq ROWSUM(A, r_{i+1}; 1, \beta)$
(iii) Set i to 0.
    For $j = 1, 2, \cdots, \beta$:
        Do (number of nodes in column j of G) $- \alpha$ times:
            Set $b_{r_i, j}$ to $b_{r_i, j} + 1$.
            Set i to $(i+1) \bmod \tilde{\alpha}$.

Figure 3.2(a) gives an example of matrices A, B, and C for the (5, 6)-jagged grid shown in Fig. 3.1(a).

For $1 \leqq i \leqq \tilde{\alpha}$, $1 \leqq j, j' \leqq \beta$, and $1 \leqq q+1, q'+1, q+k, q'+k \leqq \beta$, note the following properties of matrix A:

(PA1) $a_{i,j} \in \{1, 2\}$ (from Lemma 2.1.2)
(PA2) $COLSUM(A, j; 1, \tilde{\alpha}) = \alpha$ (from Corollary 2.1.1)
(PA3) $|COLSUM(A, j; 1, i) - COLSUM(A, j'; 1, i)| \leqq 1$ (from Lemma 2.1.2)
(PA4) $|ROWSUM(A, i; q+1, q+k) - ROWSUM(A, i'; q'+1, q'+k)| \leqq 1$ (see Appendix A).

For $1 \leqq i \leqq \tilde{\alpha}$, $1 \leqq j, j' \leqq \beta$, and $1 \leqq q+1, q'+1, q+k, q'+k \leqq \beta$, properties of B include:

(PB1) Since $\tilde{\alpha} \geqq 2$, $b_{i,j} \in \{0, 1, 2\}$ (from construction)
(When $\tilde{\alpha} \geqq 3$, $b_{i,j} \in \{0, 1\}$, but when $\tilde{\alpha} = 2$, $b_{i,j}$ might be 2.)
(PB2) $COLSUM(B, j; 1, \tilde{\alpha}) =$ number of nodes in column j of jagged grid $G - \alpha \leqq 3$ (by construction)
(PB3) $|COLSUM(B, j; 1, i) - COLSUM(B, j'; 1, i)| \leqq 3$ (from PB2)
(PB4) $|ROWSUM(B, i; q+1, q+k) - ROWSUM(B, i'; q'+1, q+k)| \leqq 2$ (see Appendix B).

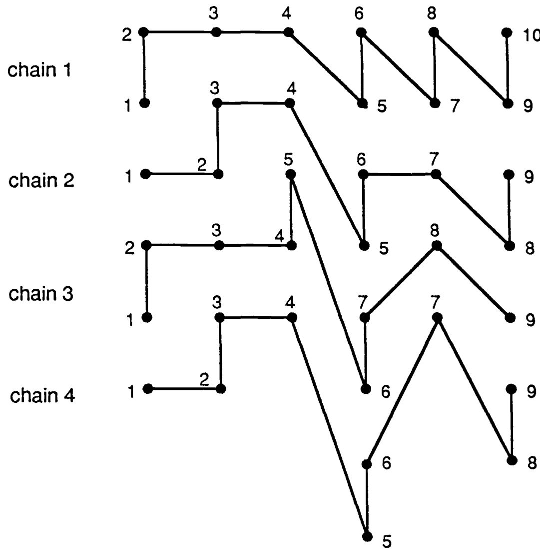
For $1 \leqq i \leqq \tilde{\alpha}$, $1 \leqq j, j' \leqq \beta$, and $1 \leqq q+1, q'+1, q+k, q'+k \leqq \beta$, properties of $C = A + B$, which follow from the properties of A and B and how B is constructed, include:

(PC1) $c_{i,j} \in \{1, 2, 3, 4\}$
(PC2) $COLSUM(C, j; 1, \tilde{\alpha}) =$ number of nodes in column j of jagged grid G
(PC3) $|COLSUM(C, j; 1, i) - COLSUM(C, j'; 1, i)| \leqq 4$
(PC4) $|ROWSUM(C, i; q+1, q+k) - ROWSUM(C, i'; q'+1, q'+k)| \leqq 3$
(PC5) $ROWSUM(C, i; 1, \beta) \in \{\lfloor n/\tilde{\alpha} \rfloor, \lceil n/\tilde{\alpha} \rceil\}$.

The $\tilde{\alpha}$ chains are drawn in G according to the $\tilde{\alpha} \times \beta$ matrix C: the ith chain is comprised of $c_{i,j}$ nodes from column j. In particular, the first $c_{1,j}$ nodes of column j will belong to chain 1, the next $c_{2,j}$ nodes of column j will belong to chain 2, the next $c_{3,j}$ nodes of column j will belong to chain 3, and so on. Figure 3.2(b) shows the resulting partition of the (5, 6)-jagged grid shown in Fig. 3.1(a) into four chains. Once again, we use line segments to join together nodes in the same chain, with the convention

$$A = \begin{bmatrix} 2&1&1&1&2&1 \\ 1&2&1&1&1&2 \\ 1&1&2&1&1&1 \\ 1&1&1&2&1&1 \end{bmatrix} \qquad B = \begin{bmatrix} 0&0&0&1&0&1 \\ 0&0&0&1&0&0 \\ 1&0&0&1&0&0 \\ 0&1&0&0&0&1 \end{bmatrix} \qquad C = \begin{bmatrix} 2&1&1&2&2&2 \\ 1&2&1&2&1&2 \\ 2&1&2&2&1&1 \\ 1&2&1&2&1&2 \end{bmatrix}$$

(a)



(b)

FIG. 3.2. *Matrices A, B, C, and chains for a* $(5, 6)$*-jagged grid.*

that adjacent nodes in the same column belonging to the same chain are joined by a line segment, and the topmost node in column j belonging to chain i is joined to the bottommost node in column $j+1$ belonging to chain i. Moreover, we can number the nodes of each chain sequentially, starting at 1, proceeding along the line segments. A node numbered p in chain i is deemed to be the pth node of chain i. □

DEFINITION. A *k-column chain segment S* is a segment of a chain which spans k consecutive columns of the grid, and $size(S)$ is the number of nodes in the chain segment. □

LEMMA 3.1.1. *Using the above procedure, the n nodes of an* (α, β)*-jagged grid G can be partitioned into* $\tilde{\alpha} = 2^{\lfloor \log_2 \alpha \rfloor} \geqq 2$ *chains so that each chain contains either* $\lfloor n/\tilde{\alpha} \rfloor$ *or* $\lceil n/\tilde{\alpha} \rceil$ *nodes, and for any two k-column chain segments S and S'*, $|size(S) - size(S')| \leqq 3$. *Note that S and S' may be in different chains.*

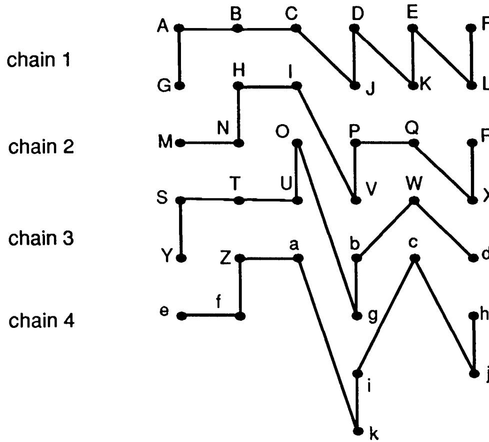
Proof. The proof follows from (PC4) and (PC5). □

We shall now provide some insight into our embedding strategy by considering our procedure for decomposing jagged grids into smaller jagged grids in the context

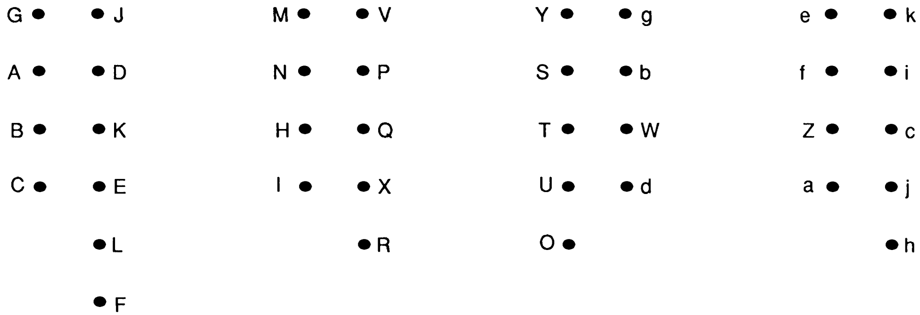
of trying to embed a three-dimensional grid into its optimal hypercube. Assume that you have an $(\alpha, k\gamma)$-jagged grid corresponding to a three-dimensional $\alpha \times k \times \gamma$ grid such that node $[x_1, x_2, x_3]$ of the three-dimensional grid is mapped to the (x_1)th row-position of column $x_2+(x_3-1)k$ in the jagged grid. Actually, each column of this $(\alpha, k\gamma)$-jagged grid has exactly α nodes. Nodes $[x_1, x_2, x_3]$ and $[x_1+1, x_2, x_3]$ of the three-dimensional grid are said to be *first-coordinate* grid-neighbors; likewise, nodes $[x_1, x_2, x_3]$ and $[x_1, x_2+1, x_3]$ and nodes $[x_1, x_2, x_3]$ and $[x_1, x_2, x_3+1]$ are said to be *second-coordinate* and *third-coordinate* grid-neighbors. Notice that first-coordinate grid-neighbors are in adjacent row-positions within the same column of the $(\alpha, k\gamma)$-jagged grid and second-coordinate grid-neighbors are in adjacent columns in the $(\alpha, k\gamma)$-jagged grid. First of all, the $(\alpha, k\gamma)$-jagged grid is partitioned into $\tilde{\alpha}=2^{\lceil \log_2 \alpha \rfloor}$ chains of about equal size in such a way that three-dimensional grid nodes mapped to nearby row-positions (though they may be in different columns) in the $(\alpha, k\gamma)$-jagged grid will end up assigned to nearby chains (to be proved in Lemma 3.1.4). The chain to which a node is assigned determines the first $\lfloor \log_2 \alpha \rfloor$ bits of its label. This will allow grid-neighbors to have similar first $\lfloor \log_2 \alpha \rfloor$ bits. A smaller jagged grid is then constructed from each chain. Each chain is partitioned into γ k-column chain segments and each such chain segment forms a column of the smaller jagged grid. Thus, the smaller jagged grid has γ columns. Three-dimensional grid nodes mapped to nearby columns of the $(\alpha, k\gamma)$-jagged grid will be placed in nearby row-positions in the smaller (t, γ)-jagged grids though they may be placed in different columns and/or different smaller jagged grids (to be proved in Lemma 3.1.5). These smaller jagged grids also bring third-coordinate grid-neighbors somewhat closer together for further partitioning. When the $(\alpha, k\gamma)$-jagged grid is decomposed into $\tilde{\alpha}$ smaller (t, γ)-jagged grids, note that first-coordinate and second-coordinate grid-neighbors end up in the same column and third-coordinate grid-neighbors end up in adjacent columns of the smaller jagged grids, though they may be in different jagged grids. In fact, node $[x_1, x_2, x_3]$ will be placed in the x_3th column of a smaller jagged grid. The formal procedure for constructing smaller jagged grids is described as follows. The smaller jagged grids will be shown to carry certain properties which will allow further recursive decomposition.

Procedure for constructing smaller jagged grids. We decompose an $(\alpha, k\gamma)$-jagged grid into $\tilde{\alpha}=2^{\lfloor \log_2 \alpha \rfloor}(t, \gamma)$-jagged grids in the following manner, where t is the size of the smallest k-column chain segment of G. After drawing $\tilde{\alpha}$ chains within an $(\alpha, k\gamma)$-jagged grid G using the above procedure, we can construct a (t, γ)-jagged grid G' from each chain by simply mapping the nodes of the k-column chain segment spanning columns 1 to k to the first column of G', the nodes of the k-column chain segment spanning columns $k+1$ to $2k$ to the second column, the nodes of the k-column chain segment spanning columns $2k+1$ to $3k$ to the third column, and so on. In particular, the pth node of the chain will be the $((p-1) \bmod k+1)$st node of a column in G', and a node in column j of G will be mapped to column $\lceil j/k \rceil$ of G'. The (5, 6)-jagged grid of Fig. 3.3(a) can be decomposed into the four (4, 2)-jagged grids shown in Fig. 3.3(b) using this procedure with $k=3$. It can also be decomposed into the four (2, 3)-jagged grids shown in Fig. 3.3(c) using this procedure with $k=2$. The nodes in Fig. 3.3(a) have been labeled with letters, and the nodes of the smaller jagged grids in Fig. 3.3(b) and 3.3(c) have also been labeled to reflect their origin within the jagged grid in Fig. 3.3(a). □

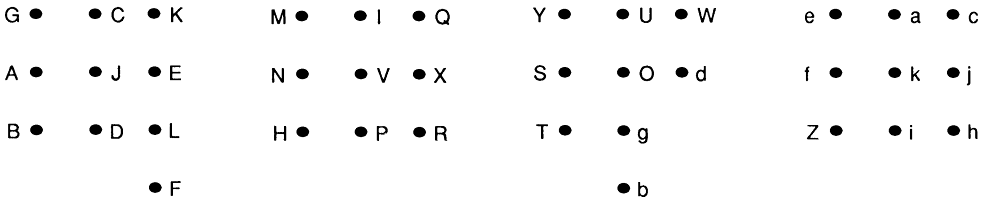
The following two lemmas show that the newly-constructed jagged grids from two compatible jagged grids are always compatible if they are decomposed in the same fashion.



(a)



(b)



(c)

FIG. 3.3. *Example of decomposing into smaller jagged grids.*

LEMMA 3.1.2. *Using the above procedure, the n nodes of an $(\alpha, k\gamma)$-jagged grid G, $k \geqq 2$, can be mapped to $\tilde{\alpha} = 2^{\lfloor \log_2 \alpha \rfloor}$ pairwise compatible (t, γ)-jagged grids, each grid containing either $\lfloor n/\tilde{\alpha} \rfloor$ or $\lceil n/\tilde{\alpha} \rceil$ nodes, where t is the size of the smallest k-column segment of G.*

Proof. With t defined as in the lemma and the property that any two k-column chain segments of G differ by at most 3 in size (see Lemma 3.1.1), as a column in G' is taken from a k-column chain segment of G, we are ensured that each column of G'

will have at least t and at most $t+3$ nodes. Clearly, G' will have γ columns, and with $k \geqq 2$, $t \geqq 2$. Any q-column slice of G' is created from a kq-column chain segment of G, hence any two q-column slices of G' will differ by at most 3 in size (see Lemma 3.1.1). In short, each jagged grid constructed is faithful to the definition of (t, γ)-jagged grids.

Since each chain has either $\lfloor n/\tilde{\alpha} \rfloor$ or $\lceil n/\tilde{\alpha} \rceil$ nodes (see Lemma 3.1.1), each jagged grid will have either $\lfloor n/\tilde{\alpha} \rfloor$ or $\lceil n/\tilde{\alpha} \rceil$ nodes. Furthermore, if G_1' and G_2' are two (t, γ)-jagged grids created from G, any q-column slice of G_1' and any q-column slice of G_2' will differ by at most 3 in size, since each slice is created from the nodes of a kq-column chain segment of G (see Lemma 3.1.1). In short, the $\tilde{\alpha}$ (t, γ)-jagged grids are pairwise compatible. □

LEMMA 3.1.3. *If G and G' are two compatible $(\alpha, k\gamma)$-jagged grids, then the $2\tilde{\alpha}$ (t, γ)-jagged grids produced by the above procedure from G and G' are also pairwise compatible with each other.*

Proof. Suppose G has n nodes and G' has $n' \geqq n$ nodes. The jagged grids produced from G will have at least $\lfloor n/\tilde{\alpha} \rfloor$ nodes, while the jagged grids produced from G' will have at most $\lceil n'/\tilde{\alpha} \rceil$ nodes. Since G and G' are compatible, $n'-n \leqq 1$. So, $\lceil n'/\tilde{\alpha} \rceil - \lfloor n/\tilde{\alpha} \rfloor \leqq 1$. Hence, the $2\tilde{\alpha}$ jagged grids produced from G and G' will differ by at most 1 in size.

To argue that any q-column slice of a jagged grid created from G and any q-column slice of a jagged grid created from G' will differ by at most 3 in size, we need only argue that any kq-column chain segment of G and any kq-column chain segment of G' will differ by at most 3 in size. Suppose the chains of G and G' are defined by $C = A + B$ and $C' = A' + B'$, respectively. Note that $A = A' = A(\alpha, \beta)$. Hence,

$$|ROWSUM(A, i; p+1, p+kq) - ROWSUM(A', i'; p'+1, p'+kq)| \leqq 1 \quad \text{(from PA4)}.$$

Because G and G' are compatible, any kq-column slice from G and any kq-column slice from G' will differ by at most 3 in size, hence, in regards to matrix B and B', using a proof similar to the one in Appendix B,

$$|ROWSUM(B, i; p+1, p+kq) - ROWSUM(B', i'; p'+1, p'+kq)| \leqq 2.$$

Thus,

$$|ROWSUM(C, i; p+1, p+kq) - ROWSUM(C', i'; p'+1, p'+kq)| \leqq 3. \qquad \square$$

DEFINITION. Let $Chain[G, x, y]$ denote the chain to which the xth node of column y in jagged grid G is assigned. Let $Pos[G, x, y]$ be such that the node in row-position x of column y in G, after decomposing G into smaller jagged grids using our procedure, ends up in row-position $Pos[G, x, y]$ within one of the columns of one of the smaller jagged grids. □

For example, let G be the $(5, 6)$-jagged grid shown in Fig. 3.3(a) which was decomposed into the four $(4, 2)$-jagged grid shown in Fig. 3.3(b). Then, $Chain[G, 2, 3] = 2$ and $Pos[G, 2, 3] = 4$, i.e., the node in row-position 2 of column 3 of G belongs to chain 2 and is mapped to row-position 4 of one of the columns (in fact, column 1) in one of the $(4, 2)$-jagged grids.

Intuitively, since chains somewhat horizontally partition the nodes of a jagged grid, we can roughly deduce the chain to which a node belongs given its row-position within a column. The following lemma gives a bound on the chains to which nodes in two compatible jagged grids belong. Note that the bound is a function of their row-positions and is independent of their columns.

LEMMA 3.1.4. *Let G and G' be two compatible (α, β)-jagged grids. Then,*

$$|Chain[G, x, y] - Chain[G', x', y']| \leqq 4 + |x - x'|.$$

Proof. Let $C = A + B$ and $C' = A' + B'$, respectively, be the partitioning matrices used to draw chains in G and G' by our procedure for constructing chains. Since $A = A' = A(\alpha, \beta)$,

$$|COLSUM(A, j; 1, p) - COLSUM(A', j'; 1, p)| \leqq 1 \quad \text{(from PA3)}.$$

Since

$$0 \leqq COLSUM(B, j; 1, p), COLSUM(B', j'; 1, p) \leqq 3 \quad \text{(from PA2)},$$
$$|COLSUM(B, j; 1, p) - COLSUM(B', j'; 1, p)| \leqq 3.$$

Thus,

$$|COLSUM(C, j; 1, p) - COLSUM(C', j'; 1, p)| \leqq 4 \quad (*).$$

Suppose $z = Chain[G, x, y]$.

$$\begin{aligned} COLSUM(C', y'; 1, z-1) - 4 + x' - x &\leqq COLSUM(C, y; 1, z-1) + x' - x \quad (\text{by } (*)) \\ &< x + x' - x = x' \leqq COLSUM(C, y; 1, z) + x' - x \quad (\text{since } z = Chain[G, x, y]) \\ &\leqq COLSUM(C', y'; 1, z) + 4 + x' - x \quad (\text{by } (*)). \end{aligned}$$

In order to show that

$$z - 4 - |x - x'| \leqq Chain[G', x', y'] \leqq z + 4 + |x - x'|,$$

we need only consider the case when $z - 4 - |x - x'| \geqq 1$ and $z + 4 + |x - x'| \leqq \tilde{\alpha}$. Since $c'_{i,j} \geqq 1$,

$$COLSUM(C', y'; 1, z - 5 - |x - x'|) \leqq COLSUM(C', y'; 1, z-1) - 4 + x' - x < x'$$

and

$$x' \leqq COLSUM(C', y'; 1, z) + 4 + x' - x \leqq COLSUM(C', y'; 1, z + 4 + |x - x'|).$$

Thus the lemma follows. □

In our procedure for decomposing a jagged grid into smaller jagged grids, we take a chain from the larger original jagged grid, cut it into segments which we pull taut and turn by 90 degrees, and paste it together to form a smaller new jagged grid. Thus, the row-position of a node in the smaller new jagged grid is roughly determined by its column in the larger jagged grid. The following lemma gives a bound on the row-positions to which nodes in compatible jagged grids are assigned in the smaller new jagged grids. Note that the bound is a function of their columns in the larger jagged grid and is independent of their row-positions.

LEMMA 3.1.5. *Let G and G' be two compatible $(\alpha, k\gamma)$-jagged grids which are decomposed into smaller (t, γ)-jagged grids. Then,*

$$|Pos[G, x, y] - Pos[G', x', y']| \leqq 6 + 4|(y-1) \bmod k - (y'-1) \bmod k|.$$

Proof. Let $C = A + B$ and $C' = A' + B'$, respectively, be the partitioning matrices used to define chains in G and G' by our procedure for constructing chains. Since $A = A' = A(\alpha, k\gamma)$,

$$|ROWSUM(A, i; p+1, p+q) - ROWSUM(A', i'; p'+1, p'+q)| \leqq 1 \quad \text{(from PA4)}.$$

Since any q-column slice from G and any q-column slice from G' will differ by at most 3 in size, by a proof similar to the one used in Appendix B, we have

$$|ROWSUM(B, i; p+1, p+q) - ROWSUM(B', i'; p'+1, p'+q)| \leqq 2.$$

Thus

$$|ROWSUM(C, i; p+1, p+q) - ROWSUM(C', i'; p'+1, p'+q)| \leqq 3 \quad (**).$$

Let $y = rk + s$ and $y' = r'k + s'$ where $1 \leqq s, s' \leqq k$, and r, r', s, s' are integers. Without loss of generality, suppose that $s \leqq s'$. Then, observe that, since $1 \leqq c_{i,j} \leqq 4$ and from the way the smaller jagged grids are constructed, we have

$$\begin{aligned}1 + ROWSUM(C, Chain[G, x, y]; rk+1, rk+s-1)\\ \leqq Pos[G, x, y]\\ \leqq 4 + ROWSUM(C, Chain[G, x, y]; rk+1, rk+s-1).\end{aligned}$$

As $c'_{i,j} \geqq 1$,

$$\begin{aligned}(s'-s+1) + ROWSUM(C', Chain[G', x', y']; r'k+1, r'k+s-1)\\ \leqq 1 + ROWSUM(C', Chain[G', x', y']; r'k+1, r'k+s'-1)\\ \leqq Pos[G', x', y'].\end{aligned}$$

As $c'_{i,j} \leqq 4$, similarly,

$$\begin{aligned}Pos[G', x', y'] &\leqq 4 + ROWSUM(C', Chain[G', x', y']; r'k+1, r'k+s'-1)\\ &\leqq 4(s'-s+1) + ROWSUM(C', Chain[G', x', y']; r'k+1, r'k+s-1).\end{aligned}$$

Because of (**), $|Pos[G, x, y] - Pos[G', x', y']| \leqq 6 + 4(s'-s)$. □

3.2. The embedding strategy. To embed the d-dimensional $n_1 \times n_2 \times \cdots \times n_d$ grid X, $n_i \geqq 2$ for $i = 1, 2, \cdots, d$, into its optimal hypercube, we generalize the method previously described for the three-dimensional grid and use the following procedure.

THE STRATEGY.

Step 0: First map X to an $(n_1, n_2 n_3 \cdots n_d)$-jagged grid G by mapping node $[x_1, x_2, \cdots, x_d]$ of X, where $1 \leqq x_i \leqq n_i$ for $i = 1, 2, \cdots, d$, to the (x_1)th node of column $x_2 + (x_3-1)n_2 + (x_4-1)n_2 n_3 + \cdots + (x_d-1)n_2 n_3 \cdots n_{d-1}$ of jagged grid G. Let $t_1 = n_1$.

Steps $\mathbf{p = 1, 2, \cdots, d-1}$: Decompose each of the $(t_p, n_{p+1} n_{p+2} \cdots n_d)$-jagged grids created by the previous step into $\tilde{t}_p = 2^{\lfloor \log_2 t_p \rfloor} (t_{p+1}, n_{p+2} n_{p+3} \cdots n_d)$-jagged grids, using the procedure described in § 3.1. Let $G_{i_1, i_2, \cdots i_{p-1}, i_p}$ denote the jagged grid constructed from the (i_p)th chain of $G_{i_1, i_2, \cdots i_{p-1}}$ when $p > 1$, or G when $p = 1$.

Step d: We now have the nodes of grid X partitioned into $\tilde{t}_1 \tilde{t}_2 \cdots \tilde{t}_{d-1}$ $(t_d, 1)$-jagged grids with t_d or $t_d + 1$ nodes each (because they are all pairwise compatible). Map the (i_d)th node (of column 1) in the $(t_d, 1)$-jagged grid $G_{i_1, i_2, \cdots, i_{d-1}}$ to the hypercube node labeled with the concatenation of

$$GRAY(\lfloor \log_2 t_1 \rfloor, i_1), GRAY(\lfloor \log_2 t_2 \rfloor, i_2), \cdots, GRAY(\lfloor \log_2 t_{d-1} \rfloor, i_{d-1})$$

and $GRAY(\lceil \log_2 (t_d+1) \rceil, i_d)$, where $GRAY$ is as defined in § 2. □

Through the above strategy, the nodes of the d-dimensional grid X are ultimately partitioned into power-of-two groups of about the same number of nodes each, in particular, into $\tilde{t}_1 \tilde{t}_2 \cdots \tilde{t}_{d-1}$ $(t_d, 1)$-jagged grids with either t_d or $t_d + 1$ nodes per jagged grid. Each node is given a unique label since nodes belonging to different $(t_d, 1)$-jagged grids will receive different initial $\lfloor \log_2 t_1 \rfloor + \cdots + \lfloor \log_2 t_{d-1} \rfloor$ bits for their labels and nodes belonging to the same $(t_d, 1)$-jagged grid will receive different final $\lceil \log_2 (t_d+1) \rceil$ bits for their labels. Since $\tilde{t}_i$'s are powers of two,

$$\tilde{t}_1 \tilde{t}_2 \cdots \tilde{t}_{d-1} t_d \leqq n_1 n_2 \cdots n_d \leqq \tilde{t}_1 \tilde{t}_2 \cdots \tilde{t}_{d-1}(t_d+1),$$

we will indeed end up with labels of

$$\lceil \log_2 n_1 n_2 \cdots n_d \rceil = \log_2 \tilde{t}_1 + \log_2 \tilde{t}_2 + \cdots + \log_2 \tilde{t}_{d-1} + \lceil \log_2 (t_d + 1) \rceil$$
$$= \lfloor \log_2 t_1 \rfloor + \lfloor \log_2 t_2 \rfloor + \cdots + \lfloor \log_2 t_{d-1} \rfloor + \lceil \log_2 (t_d + 1) \rceil$$

bits. Thus, we are indeed mapping into X's optimal hypercube. Now, let us turn our attention to the dilation involved.

The following lemma shows that for any two grid-neighbors, their corresponding column, chain, and row-position numbers in the newly-constructed jagged grid at each stage of the construction differs only by a constant amount.

LEMMA 3.2.1. *Let U and V denote, respectively, the nodes $(x_1, \cdots, x_{j-1}, x_j, x_{j+1}, \cdots, x_d)$ and $(x_1, \cdots, x_{j-1}, x_j + 1, x_{j+1}, \cdots, x_d)$ of grid X, i.e., U and V are grid-neighbors. Let $COLUMN(p, U)$, $CHAIN(p, U)$, and $POS(p, U)$ denote, respectively, the column in the $(t_{p+1}, n_{p+2} n_{p+3} \cdots n_d)$-jagged grid, the chain in the $(t_{p+1}, n_{p+2} n_{p+3} \cdots n_d)$-jagged grid, and the row-position in the $(t_{p+2}, n_{p+3} n_{p+4} \cdots n_d)$-jagged grid to which node U is mapped. Then,*

$$|(COLUMN(p, U) - 1) \bmod n_{p+2} - (COLUMN(p, V) - 1) \bmod n_{p+2}| = \begin{cases} 1 & \text{if } p = j-2 \\ 0 & \text{otherwise} \end{cases},$$

$$\begin{aligned} |POS(p, U) - POS(p, V)| &\leqq 6 + 4 |(COLUMN(p, U) - 1) \bmod n_{p+2} \\ &\qquad - (COLUMN(p, V) - 1) \bmod n_{p+2}| \\ &\leqq \begin{cases} 10 & \text{if } p = j-2 \\ 6 & \text{otherwise} \end{cases}, \end{aligned}$$

$$\begin{aligned} |CHAIN(p, U) - CHAIN(p, V)| &\leqq 4 + |POS(p-1, U) - POS(p-1, V)| \\ &\leqq \begin{cases} 14 & \text{if } p = j-1 \\ 10 & \text{otherwise} \end{cases}. \end{aligned}$$

Proof. Note that

$$COLUMN(0, U) = x_2 + (x_3 - 1) n_2 + (x_4 - 1) n_2 n_3 + \cdots + (x_d - 1) n_2 n_3 \cdots n_{d-1}$$

and

$$COLUMN(0, V) = \begin{cases} COLUMN(0, U) + n_2 \cdots n_{j-1} & \text{if } j > 2 \\ COLUMN(0, U) + 1 & \text{if } j = 2. \\ COLUMN(0, U) & \text{if } j < 2 \end{cases}$$

Since a node in column j of the $(t_{p+1}, n_{p+2} n_{p+3} \cdots n_d)$-jagged grid will be mapped to column $\lceil j / n_{p+2} \rceil$ of a $(t_{p+2}, n_{p+3} n_{p+4} \cdots n_d)$-jagged grid, then

$$COLUMN(p, U) = x_{p+2} + (x_{p+3} - 1) n_{p+2} + \cdots + (x_d - 1) n_{p+2} n_{p+3} \cdots n_{d-1}$$

and

$$COLUMN(p, V) = \begin{cases} COLUMN(p, U) + n_{p+2} \cdots n_{j-1} & \text{if } j > p+2 \\ COLUMN(p, U) + 1 & \text{if } j = p+2. \\ COLUMN(p, U) & \text{if } j < p+2 \end{cases}$$

The first part of the lemma readily follows. Applying Lemmas 3.1.5 and 3.1.4, the rest of the lemma also follows. □

LEMMA 3.2.2. *Let U and V be two neighbors in the d-dimensional grid X, and $DILATION(U, V)$ denote the number of edges in the hypercube by which U and V are distanced. Since binary-reflected Gray code has the property that $GRAY(t, i)$ and $GRAY(t, i+j)$ will differ by less than or equal to $\lceil \log_2(3j/2) \rceil$ bits* [MS],

$$\begin{aligned} DILATION(U, V) &\leqq \lceil \log_2(3|POS(d-2, U) - POS(d-2, V)|/2) \rceil \\ &\quad + \sum_{p=0}^{d-2} \lceil \log_2(3|CHAIN(p, U) - CHAIN(p, V)|/2) \rceil \\ &\leqq 4d+1 = O(d). \end{aligned}$$

THEOREM 3.1. *Any d-dimensional grid can be embedded into its optimal hypercube with a dilation of $O(d)$.*

4. Concluding remarks. We have successfully solved the problem of embedding all two-dimensional grids into their optimal hypercubes with dilation at most 2 and have given an $O(d)$ upper bound on dilation for embedding d-dimensional grids into their optimal hypercubes. The d-dimensional strategy actually grew from the technique we developed for handling three-dimensional grids. That technique and its analysis [C2], being especially tailored to the three-dimensional case, results in dilation 7 rather than 13 ($=4d+1$) for embedding three-dimensional grids into their optimal hypercubes. Insofar as lower bounds are concerned, we only know that not all grids are subgraphs of their optimal hypercubes, implying that dilation at least 2 is necessary; nothing more is known.

Appendix A: Proof of property (PA4).

$$\begin{aligned} ROWSUM(A, i; q+1, q+k) &= \sum_{j=q+1}^{q+k} a_{i,j} = \sum_{j-q+1}^{q+k} a_{(i-j) \bmod \tilde{\alpha}+1,1} = \sum_{j=1}^{k} a_{(i-q-j) \bmod \tilde{\alpha}+1,1} \\ &= \sum_{j=1}^{k} a_{(I-j) \bmod \tilde{\alpha}+1,1} = ROWSUM(A, I; 1, k) \end{aligned}$$

where

$$I = (i-q) \bmod \tilde{\alpha}.$$

From Lemma 2.1.3, we have $\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq ROWSUM(A, i; q+1, q+k) \leqq \lceil k\alpha/\tilde{\alpha} \rceil$.

Similarly, we have $\lfloor k\alpha/\tilde{\alpha} \rfloor \leqq ROWSUM(A, i'; q'+1, q'+k) \leqq \lceil k\alpha/\tilde{\alpha} \rceil$.

Thus, $|ROWSUM(A, i; q+1, q+k) - ROWSUM(A, i'; q'+1, q'+k)| \leqq 1$. □

Appendix B: Proof of property (PB4). Let S and S' be two k-column slices from columns $q+1$ and $q+k$ and columns $q'+1$ to $q'+k$, respectively. By the definition of jagged grids, $|size(S) - size(S')| \leqq 3$. From the algorithm constructing matrix B, the extra elements $m = size(S) - k\alpha$ and $m' = size(S') - k\alpha$ in these two k-column slices are evenly distributed among the $\tilde{\alpha}$ rows in B. Thus, we have

$$ROWSUM(B, i; q+1, q+k) \in \{\lfloor m/\tilde{\alpha} \rfloor, \lfloor m/\tilde{\alpha} \rfloor\},$$

$$ROWSUM(B, i'; q'+1, q'+k) \in \{\lfloor m'/\tilde{\alpha} \rfloor, \lfloor m'/\tilde{\alpha} \rfloor\}.$$

As $|m - m'| \leqq 3$ and $\tilde{\alpha} \geqq 2$, we thus have

$$|ROWSUM(B, i; q+1, q+k) - ROWSUM(B, i'; q'+1, q'+k)| \leqq 2. \quad \square$$

Acknowledgments. I would like to express my thanks to Francis Chin, Hal Sudborough, Tom Leighton, Hongfei Liu, Sheshu Madhavapeddy, and Joel Lee for their interest and help in looking over the many drafts of this paper.

REFERENCES

[BMS] S. Bettayeb, Z. Miller, and I. H. Sudborough, *Embedding grids into hypercubes*, in Proc. 3rd Aegean Workshop on Computing, Patras, Greece, 1988.

[BS] J. E. Brandenburg and D. S. Scott, *Embeddings of communication trees and grids into hypercubes*, Intel Scientific Computers Report #280182-001, Intel Scientific Computers, CA, 1985.

[C1] M. Y. Chan, *Dilation-2 embedding of grids into hypercubes*, in Proc. International Conference on Parallel Processing, St. Charles, IL, August 1988.

[C2] ———, *Embedding of 3-dimensional grids into optimal hypercubes*, in Proc. Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Monterey, CA, March 1989.

[C3] ———, *Embedding of d-dimensional grids into optimal hypercubes*, in Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, June 1989.

[CC] M. Y. Chan and F. Y. L. Chin, *On embedding rectangular grids in hypercubes*, IEEE Trans. Comput., C-37 (1988), pp. 1285–1288.

[G] D. S. Greenberg, *Optimal expansion embeddings of meshes in hypercubes*, Tech. Report YALEU/CSD/RR-535, Department of Computer Science, Yale University, New Haven, CT, August 1987.

[HJ] C. T. Ho and S. L. Johnsson, *On the embedding of arbitrary meshes in Boolean cubes with expansion two dilation two*, in Proc. International Conference on Parallel Processing, St. Charles, IL, August 1987.

[LS] M. Livingston and Q. F. Stout, *Embeddings in hypercubes*, in Proc. Sixth International Conference on Mathematical Modeling, Washington State University, Pullman, WA, August 1987.

[MS] S. Madhavapeddy and I. H. Sudborough, *A note on cyclic binary codes*, Tech. Report UTDCS-6-89, Computer Science Department, University of Texas, Dallas, TX, March 1989.

[SS] Y. Saad and M. H. Schultz, *Topological properties of hypercubes*, Res. Report 389, Department of Computer Science, Yale University, New Haven, CT, June 1985.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 865-877, October 1991
© 1991 Society for Industrial and Applied Mathematics
004

PP IS AS HARD AS THE POLYNOMIAL-TIME HIERARCHY*

SEINOSUKE TODA†

Abstract. In this paper, two interesting complexity classes, PP and $\oplus$P, are compared with PH, the polynomial-time hierarchy. It is shown that every set in PH is polynomial-time Turing reducible to a set in PP, and PH is included in BP · $\oplus$P. As a consequence of the results, it follows that PP $\subseteq$ PH (or $\oplus$P $\subseteq$ PH) implies a collapse of PH. A stronger result is also shown: every set in PP(PH) is polynomial-time Turing reducible to a set in PP.

Key words. polynomial-time hierarchy, probabilistic Turing machine, polynomial-time Turing reductions, parity, randomized reduction

AMS(MOS) subject classifications. 68Q15, 03D15

1. Introduction. Since the notion of probabilistic Turing machines was introduced by Gill [5], much attention has been given to several questions about its computational power. One of those questions is whether PP is more powerful than PH (the polynomial-time hierarchy), where PP denotes the class of sets accepted by polynomial-time-bounded probabilistic Turing machines with two-sided unbounded error probability. In particular, it is important in the theory of computational complexity to ask whether PH is included in PP, or to ask whether all sets in PH are reducible to sets in PP under a suitable reducibility. This has been an open question discussed in many papers [1], [2], [10], [12], [15], [16], [19-21]. It was shown by Gill [5] that NP $\cup$ co-NP is included in PP. It is not known, however, whether Δ_2^P is included in PP. For this question, Beigel, Hemachandra, and Wechsung [3] have recently shown that $P^{NP[\log]}$ is included in PP. This is the strongest result known currently for the containment question of PH in PP. Some related results have been shown in [20].

In this paper, we give an affirmative answer to one of the above questions. We show that all sets in PH are $\leq_T^P$-reducible to a set in PP. Namely, our Main Theorem in this paper is stated as follows.

MAIN THEOREM. PH $\subseteq$ P(PP)

As an immediate consequence, we see that PP is not included in PH unless PH collapses to a finite level. This gives us evidence that PP is harder than PH. In the process of proving the Main Theorem, we show an interesting result about the hardness of the class $\oplus$P. This class was introduced by Papadimitriou and Zachos [13] and further investigated in several papers [13], [25], [15]. We show that all sets in PH are reducible to a set in this class under polynomial-time randomized reductions with two-sided bounded error probability. It was shown by Valiant and Vazirani [25] that all sets in NP are reducible to a set in $\oplus$P under polynomial-time randomized reductions with one-sided bounded error probability. These randomized reductions are stronger, but in other respects our result extends theirs. In fact, they asked how computationally difficult $\oplus$P is. Our result is an answer to their open question.

Our proof of the main theorem proceeds as follows. In § 3, we show that PH is included in BP · $\oplus$P, where BP · denotes the BP-operator introduced by Schöning [15]. Intuitively speaking, a set is in BP · $\oplus$P if and only if it is reducible to a set in $\oplus$P

* Received by the editors February 21, 1989; accepted for publication (in revised form) December 12, 1990. This research was supported in part by International Information Science Foundation grant 89 2 2 176.

† Department of Computer Science and Information Mathematics, University of Electro-communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182, Japan.

under a polynomial-time randomized reduction with two-sided bounded error probability. The proof of it is based on a result by Valiant and Vazirani [25] and a result by Schöning [15]. In § 4, we show that $\mathsf{BP} \cdot \oplus\mathrm{P}$ is included in P(PP). In fact, we will show a stronger result than this. The proof is based on a structural property of $\oplus$P discovered in this paper. At the end of § 4, we will mention a stronger result than the Main Theorem above: $\mathrm{PP(PH)} \subseteq \mathrm{P(PP)}$. This result is obtained by combining the technique in this paper with a result by Köbler et al. [10].

2. Preliminaries. We assume that the reader is familiar with the basic concepts of computational complexity theory. Let Σ be a finite alphabet. For a string $w \in \Sigma^*$, $|w|$ denotes the length of w. For a set $L \subseteq \Sigma^*$, $\bar{L}$ denotes the complement of L. For a class **K** of sets, co-**K** denotes the class of sets whose complement is in **K**. Let Σ^n (respectively, $\Sigma^{\leq n}$ and $\Sigma^{<n}$) denote the set of strings with length n (respectively, length at most n and less than n). For a finite set $X \subseteq \Sigma^*$, $\|X\|$ denotes the number of strings in X. Let **N** denote the set of natural numbers.

Our sets in this paper are over $\Sigma = \{0, 1, \#\}$ unless otherwise specified. The symbol # is usually used as a delimiter among strings of $\{0, 1\}^*$. A pairing function (respectively, a k-tuple function) over $\{0, 1\}^*$ is represented by delimiting two strings (respectively, k strings) by this symbol.

Our models of computation are variations of *polynomial-time-bounded oracle Turing machines* (deterministic, nondeterministic, or probabilistic). Our oracle machines are usual ones. For an oracle machine M and an oracle set A, M(A) denotes that M uses A as an oracle. A polynomial-time-bounded deterministic (respectively, nondeterministic) oracle machine is abbreviated by an *oracle* P-*machine* (respectively, *oracle* NP-*machine*). A polynomial-time-bounded probabilistic oracle machine with two-sided unbounded error probability (respectively, with two-sided bounded error probability) is abbreviated by an *oracle* PP-*machine* (respectively, an *oracle* BPP-*machine*). In the unrelativized cases, we omit the term "oracle." For example, an oracle NP-machine with the empty set as an oracle is simply called an NP-*machine*.

For an oracle set A, P(A) denotes the class of sets accepted by oracle P-machine with oracle A. NP(A), PP(A), and BPP(A) are defined similarly. $\oplus$P(A) denotes the class of sets L for which there exists an oracle NP-machine M such that for each x, x is in L if and only if the number of accepting computation paths of M(A) on x is odd. This class was defined by Papadimitriou and Zachos [13]. For a class **K** of oracle sets, $\mathrm{P}(\mathbf{K}) = \bigcup \{\mathrm{P}(A)\colon A \in \mathbf{K}\}$. Other classes are defined similarly. The unrelativized classes are defined by setting the oracle set to the empty set, and the specification of oracle set is omitted in this case.

We assume that all polynomial-time-bounded oracle machines M satisfy the following conditions.

(1) Its transition function has at most two possible transitions from each configuration.

(2) All computation paths of M are encoded into a string of $\{0, 1\}^*$ by the usual manner, where a computation path may contain possible answers from a given oracle, and the oracle answer "yes" (respectively, "no") is encoded by 0 (respectively, 1).

These assumptions are technical ones. Obviously, we lose no generality under these assumptions.

Let X be a finite set of strings and R be a predicate over strings. In this paper, we denote by $\mathrm{Prob}(\{w \in X\colon R(w)\})$ the probability that $R(w)$ is true for randomly chosen w from X under uniform distribution. In [15], Schöning introduced the BP-operator, which produces a probabilistic class from a given class. He also defined

$\oplus$-operator in [16], as an abstraction of the class $\oplus$P. In [26], Wagner defined the counting operator, based on a characterization of PP in [14], [26]. We give those definitions here. The following definition of the counting operator is different from the original one; however, it is easy to see that both definitions define the same concept.

DEFINITION 2.1 [15], [16], [26]. Let $\mathbf{K}$ be a class of sets and let L be a set. Then we define some new classes denoted by $\oplus \cdot \mathbf{K}$, $\mathsf{BP} \cdot \mathbf{K}$, and $\mathbf{C} \cdot \mathbf{K}$ as follows.

(1) $L \in \oplus \cdot \mathbf{K}$ if there exist a set $A \in \mathbf{K}$ and a polynomial p such that for all $x \in \Sigma^*$,

$$x \in L \leftrightarrow \|\{w \in \{0,1\}^{p(|x|)} : x \# w \in A\}\| \text{ is odd.}$$

(2) $L \in \mathsf{BP} \cdot \mathbf{K}$ if there exist a set $A \in \mathbf{K}$, a polynomial p, and a constant $\alpha > 0$ such that, for all $x \in \Sigma^*$,

$$\text{Prob}\,(\{w \in \{0,1\}^{p(|x|)} : x \# w \in A \leftrightarrow x \in L\}) \geqq \tfrac{1}{2}\alpha.$$

(3) $L \in \mathbf{C} \cdot \mathbf{K}$ if there exist a set $A \in \mathbf{K}$ and a polynomial p such that for all $x \in \Sigma^*$,

$$\text{Prob}\,(\{w \in \{0,1\}^{p(|x|)} : x \# w \in A \leftrightarrow x \in L\}) > \tfrac{1}{2}.$$

It is easy to see that $\mathbf{C} \cdot \mathrm{P} = \mathrm{PP}$, $\mathsf{BP} \cdot \mathrm{P} = \mathrm{BPP}$, and $\oplus \cdot \mathrm{P} = \oplus \mathrm{P}$. The following propositions are basic properties of the above operators, which follow immediately from the definitions. These will be used implicitly in later arguments.

PROPOSITION 2.2. *Let $\mathbf{K}_1$ and $\mathbf{K}_2$ be arbitrary classes of sets. Then, the following statements hold.*

(1) *If $\mathbf{K}_1 \subseteq \mathbf{K}_2$, then $\oplus \cdot \mathbf{K}_1 \subseteq \oplus \cdot \mathbf{K}_2$.*

(2) *If $\mathbf{K}_1$ is closed under marked union with sets of the form $\{x \# 0^{p(|x|)} : x \in \Sigma^*\}$ (for arbitrary polynomial p such sets are in* P*), then $\oplus \cdot \mathbf{K}_1$ is closed under complementation.*

(3) *If $\mathbf{K}_1$ is such that, for all sets $L \in \mathbf{K}_1$, the set $\{x \# x : x \in L\}$ also belongs to $\mathbf{K}_1$, then $\mathbf{K}_1 \subseteq \oplus \cdot \mathbf{K}_1$.*

PROPOSITION 2.3. *Let $\mathbf{K}_1$ and $\mathbf{K}_2$ be any classes of sets. Then the following statements hold.*

(1) *If $\mathbf{K}_1 \subseteq \mathbf{K}_2$, then $\mathsf{BP} \cdot \mathbf{K}_1 \subseteq \mathsf{BP} \cdot \mathbf{K}_2$.*

(2) *co-$\mathsf{BP} \cdot \mathbf{K}_1 \subseteq \mathsf{BP} \cdot$ co-$\mathbf{K}_1$. Hence, if $\mathbf{K}_1$ is closed under complementation, then co-$\mathsf{BP} \cdot \mathbf{K}_1 = \mathsf{BP} \cdot \mathbf{K}_1$.*

(3) *If $\mathbf{K}_1$ is closed under padding (i.e., $L \in \mathbf{K}_1$ implies $\{x \# y : x \in L$ and $y \in \{0,1\}^*\} \in \mathbf{K}_1$ for each set L), then $\mathbf{K}_1 \subseteq \mathsf{BP} \cdot \mathbf{K}_1$.*

PROPOSITION 2.4. *Let $\mathbf{K}_1$ and $\mathbf{K}_2$ be arbitrary classes of sets. Then the following statements hold.*

(1) *If $\mathbf{K}_1 \subseteq \mathbf{K}_2$, then $\mathbf{C} \cdot \mathbf{K}_1 \subseteq \mathbf{C} \cdot \mathbf{K}_2$.*

(2) *co-$\mathbf{C} \cdot \mathbf{K}_1 \subseteq \mathbf{C} \cdot$ co-$\mathbf{K}_1$. Hence, if $\mathbf{K}_1$ is closed under complementation, then co-$\mathbf{C} \cdot \mathbf{K}_1 = \mathbf{C} \cdot \mathbf{K}_1$.*

(3) *If $\mathbf{K}_1$ is closed under padding, then $\mathbf{K}_1 \subseteq \mathbf{C} \cdot \mathbf{K}_1$.*

(4) $\mathsf{BP} \cdot \mathbf{K}_1 \subseteq \mathbf{C} \cdot \mathbf{K}_1$.

We can easily see that all the classes to be built in this paper satisfy the closure properties mentioned in the above propositions (except possibly for complementation). For example, $\Pi_k^{\mathrm{P}} (k \geqq 0)$ is closed under taking marked union with the set of the form $\{x \# 0^{p(|x|)} : x \in \Sigma^*\}$; hence, $\oplus \cdot \Pi_k^{\mathrm{P}}$ is closed under complementation; we will use this fact in the next section.

In the later sections, we will be concerned with several reducibility notions defined below.

DEFINITION 2.5. Let A and B be arbitrary sets. A is said to be $\leq_m^{\mathrm{P}}$*-reducible* to $B (A \leq_m^{\mathrm{P}} B)$ if there exists a function f computable in polynomial time such that for each x, $x \in A$ if and only if $f(x) \in B$. A is said to be $\leq_{\mathrm{maj}}^{\mathrm{P}}$*-reducible* to $B (A \leq_{\mathrm{maj}}^{\mathrm{P}} B)$ if

there exists a function f computable in polynomial time such that for each $x, f(x) = y_1\#y_2\# \cdots \#y_m$ ($m \geqq 1$), and $x \in A$ if and only if the majority of y_i's are in B. A is said to be $\leq_T^P$-*reducible* to B ($A \leq_T^P B$) if there exists an oracle P-machine that accepts A with oracle B. Let $\mathbf{K}$ be a class of sets and let $\leq_r^P$ denote an arbitrary reducibility. A set L is $\leq_r^P$-*hard* for $\mathbf{K}$ if every set in $\mathbf{K}$ is $\leq_r^P$-reducible to L; if $L \in \mathbf{K}$ in addition, then L is said to be $\leq_r^P$-*complete* for $\mathbf{K}$. Then we say that $\mathbf{K}$ is closed under $\leq_r^p$ if and only if for all sets A and B, $A \leq_r^p B$ and $B \in \mathbf{K}$ implies $A \in \mathbf{K}$.

Observe that for all sets A and B, $A \leq_{\text{maj}}^P B$ implies $A \leq_T^P B$; and hence, if a class $\mathbf{K}$ is closed under $\leq_T^P$, then the class is closed under $\leq_{\text{maj}}^P$.

3. $\oplus$P is hard for PH under randomized reducibility. In this section, we show that $\oplus$P is hard for the polynomial-time hierarchy under polynomial-time randomized reducibility. More precisely, our main result in this section is stated as follows.

THEOREM 3.1. $\text{PH} \subseteq \mathsf{BP} \cdot \oplus \text{P}$.

Before proving this, we give an intuitive explanation of the proof to the reader. We first show that Σ_k^P is included in $\mathsf{BP} \cdot \oplus \cdot \Pi_{k-1}^P$ for each $k \geqq 1$ (see Lemma 3.3). This generalizes a result due to Valiant and Vazirani [25] in which they showed that all NP-complete sets are reducible to a set in $\oplus$P under randomized polynomial-time reducibility. Our proof technique is essentially the same as theirs. We next observe that it is possible to swap a $\oplus$-operator and a BP-operator. In particular, we show that $\oplus \cdot \mathsf{BP} \cdot \oplus \text{P} \subseteq \mathsf{BP} \cdot \oplus \cdot \oplus \text{P}$ (see Lemma 3.6). Furthermore, we observe that it is possible to reduce two consecutive BP-operators (respectively, two consecutive $\oplus$-operators) to one operator: It was shown by Papadimitriou and Zachos [13] that $\oplus\text{P}(\oplus\text{P}) = \oplus\text{P}$. This implies that $\oplus \cdot \oplus\text{P} = \oplus\text{P}$, and we also show that $\mathsf{BP} \cdot \mathsf{BP} \cdot \oplus\text{P} = \mathsf{BP} \cdot \oplus\text{P}$ (see Lemma 3.7). At the end of this section, we put all this together to prove Theorem 3.1, using an induction on the levels of the polynomial-time hierarchy.

Now we begin to show the lemmas mentioned above. Following Valiant and Vazirani [25], we shall view strings of $\{0, 1\}^n$ as n-dimensional vectors from the vector space $\text{GF}[2]^n$. We denote by $u \cdot v$ the inner product of two vectors u and v over GF[2]. In [25], they showed the following result.

THEOREM 3.2 [25]. *Let $n \geqq 1$ and let $S \subseteq \{0, 1\}^n$ be a nonempty set. Suppose $w_1, w_2, \cdots, w_n$ are randomly chosen from $\{0, 1\}^n$. Let $S_0 = S$ and let*

$$S_i = \{v \in S: v \cdot w_1 = v \cdot w_2 = \cdots = v \cdot w_i = 0\}$$

for each $1 \leqq i \leqq n$. Let $P_n(S)$ be the probability that $\|S_i\| = 1$ for some $0 \leqq i \leqq n$. Then, $P_n(S) \geqq \frac{1}{4}$.

LEMMA 3.3. *For each $k \geqq 1, \Sigma_k^P \cup \Pi_k^P \subseteq \mathsf{BP} \cdot \oplus \cdot \Pi_{k-1}^P$.*

Proof. By Propositions 2.2(2) and 2.3(2), $\mathsf{BP} \cdot \oplus \cdot \Pi_{k-1}^P$ is closed under complementation. Hence, it suffices to show that $\Sigma_k^P \subseteq \mathsf{BP} \cdot \oplus \cdot \Pi_{k-1}^P$. Let $L \in \Sigma_k^P$. Then it was shown by Stockmeyer [18] and Wrathall [28] that there exist a set $A \in \Pi_{k-1}^P$ and a polynomial p such that for every x, $x \in L$ if and only if $x \# y \in A$ for some $y \in \{0, 1\}^{p(|x|)}$. We define a set C as follows:

$$C = \{x \# w_1 w_2 \cdots w_{p(|x|)}\text{: for each } i,\ 1 \leqq i \leqq p(|x|),\ w_i \in \{0, 1\}^{p(|x|)}, \text{and for some } j,$$
$$0 \leq j \leq p(|x|),\ \|\{y \in \{0, 1\}^{p(|x|)}: x \# y \in A \wedge (\forall i \leqq j)[w_j \cdot y = 0]\}\| \text{ is odd}\}.$$

We first show that C is in $\oplus \cdot \Pi_{k-1}^P$. Since $\oplus \cdot \Pi_{k-1}^P$ is closed under complementation, it suffices to show that C's complement, $\bar{C}$, is in $\oplus \cdot \Pi_{k-1}^P$. This can be done as follows: Given arbitrary strings x and $z = w_1 w_2 \cdots w_{p(|x|)}$ such that, for each i, $1 \leqq i \leqq p(|x|)$, $w_i \in \{0, 1\}^{p(|x|)}$,

$$
\begin{aligned}
&x \# z \in \bar{C}\\
&\leftrightarrow \text{for each } j,\ 0 \leqq j \leqq p(|x|),\ \|\{y \in \{0,1\}^{p(|x|)}: x \# y \in A \wedge (\forall i \leq j)[w_j \cdot y = 0]\}\| \text{ is even}\\
&\leftrightarrow \prod_{j=0}^{p(|x|)} (\|\{y: x \# y \in A \wedge (\forall i \leq j)[w_j \cdot y = 0]\}\| + 1) \text{ is odd.}
\end{aligned}
$$

Hence we may define a set $B \in \Pi^{\mathrm{P}}_{k-1}$ and a polynomial q such that for every $x \# z$, $(|z| = p(|x|)^2)$,

$$
\|\{u \in \{0,1\}^{q(|x|)}: x \# z \# u \in B\}\| = \prod_{j=0}^{p(|x|)} (\|\{y: x \# y \in A \wedge (\forall i \leq j)[w_j \cdot y = 0]\}\| + 1).
$$

The set B is defined by

$$
\begin{aligned}
B = \{x \# w_1 \cdots w_{p(|x|)} \# a_1 y_1 a_2 y_2 \cdots a_{p(|x|)} w_{p(|x|)}&: \text{for every } j,\ 1 \leqq j \leqq p(|x|),\\
w_j \in \{0,1\}^{p(|x|)},\ a_j \in \{0,1\},\ y_j &\in \{0,1\}^{p(|x|)}, \text{ and}\\
(\forall j, 1 \leqq j \leqq p(|x|))[a_j y_j = 0^{p(|x|)+1} \vee (a_j = 1 \wedge x \# y_j &\in A \wedge (\forall i \leq j)[w_i \cdot y_j = 0])]\}.
\end{aligned}
$$

It is easy to see that $B \in \Pi^{\mathrm{P}}_{k-1}$ and that B and the polynomial $q(n) = p(n)(1 + p(n))$ satisfy the required condition; that is, the set B and the polynomial q witness $\bar{C} \in \oplus \cdot \Pi^{\mathrm{P}}_{k-1}$.

Next, we show $L \in \mathsf{BP} \cdot \oplus \cdot \Pi^{\mathrm{P}}_{k-1}$ by using the set C. Let x be a string and let $w_1, w_2, \cdots, w_{p(|x|)}$ be randomly chosen from $\{0,1\}^{p(|x|)}$. We define

$$
S_0 = \{y \in \{0,1\}^{p(|x|)}: x \# y \in A\}
$$

and

$$
S_i = \{y \in S_0: w_1 \cdot y = w_2 \cdot y = \cdots = w_i \cdot y\}
$$

for each $1 \leqq i \leqq p(|x|)$. Let $P_{p(|x|)}(S_0)$ be the probability that $\|S_i\| = 1$ for some $0 \leqq i \leqq p(|x|)$. Then it is easy to see that

$$
\text{Prob}\,(\{w_1 \cdots w_{p(|x|)} \in \{0,1\}^{p(|x|)^2}: x \# w_1 w_2 \cdots w_{p(|x|)} \in C\}) \geqq P_{p(|x|)}(S_0).
$$

Hence, from Theorem 3.2,

(1) $x \in L \to \text{Prob}\,(\{u \in \{0,1\}^{p(|x|)^2}: x \# u \in C\}) \geqq \frac{1}{4}$ and

(2) $x \notin L \to \text{Prob}\,(\{u \in \{0,1\}^{p(|x|)^2}: x \# u \in C\}) = 0$.

The probability of (2) follows from the fact that for all x, if $x \notin L$, then $x \# y \notin A$ for every $y \in \{0,1\}^{p(|x|)}$. To amplify the probability in (1), we further define a set D as follows:

$$
D = \{x \# u_1 u_2 u_3: |u_i| = p(|x|)^2 \text{ for each } i = 1,2,3 \text{ and } x \# u_i \in C \text{ for some } i = 1,2,3\}.
$$

Then we obtain that for each x,

$$
\begin{aligned}
(3)\quad x \in L \to \text{Prob}\,&(\{u_1 u_2 u_3 \in \{0,1\}^{3p(|x|)^2}: x \# u_1 u_2 u_3 \in D\})\\
&= 1 - (\text{Prob}\,(\{u \in \{0,1\}^{p(|x|)^2}: x \# u \notin C\}))^3\\
&\geqq 1 - \tfrac{27}{64} = \tfrac{1}{2} + \tfrac{5}{64} \quad \text{and}
\end{aligned}
$$

$$
(4)\quad x \notin L \to \text{Prob}\,(\{u_1 u_2 u_3 \in \{0,1\}^{3p(|x|)^2}: x \# u_1 u_2 u_3 \in D\}) = 0.
$$

By using the same argument as when showing $C \in \oplus \cdot \Pi^{\mathrm{P}}_{k-1}$, we can show $D \in \oplus \cdot \Pi^{\mathrm{P}}_{k-1}$. This implies $L \in \mathsf{BP} \cdot \oplus \cdot \Pi^{\mathrm{P}}_{k-1}$. □

It was shown by Papadimitriou and Zachos [13] that $\oplus\mathrm{P}(\oplus\mathrm{P}) = \oplus\mathrm{P}$. This implies the following theorem. For the sake of making this paper more self-contained, we provide a sketch of their proof.

THEOREM 3.4 [13]. $\oplus P(\oplus P) = \oplus P$. *Hence we have that* $\oplus \cdot \oplus P = \oplus P$ *and that* $\oplus P$ *is closed under* $\leq_{maj}^{P}$.

Proof Sketch. Let L be a set in $\oplus P(\oplus P)$. Then there exist a set $A \in \oplus P$ and an oracle NP-machine M such that for every x, $x \in L$ if and only if the number of accepting paths of $M(A)$ on input x is odd. Let M_1 be an NP-machine that witnesses $A \in \oplus P$. Then we define an NP-machine M_2 working on a given input x as follows:

(1) M_2 first guesses a computation path w of M on input x, which includes possible oracle answers to the query strings appearing in w.
(2) If w is a rejecting path of M on x, then M_2 enters a rejecting state; otherwise, it goes to the next step.
(3) Let $y_1, y_2, \cdots, y_m$ ($z_1, z_2, \cdots, z_l$) be all the query strings which appear in w and whose corresponding oracle answers in w are "yes" (respectively, "no"). Then M_2 simulates M_1 successively for each y_i and each z_j in the following manner:
(a) For each y_i, it simply simulates M_1. If M_1 enters a rejecting state, then so does M_2; otherwise, it proceeds to the next simulation.
(b) For each z_i, it nondeterministically selects one of the following processes:
(i) M_2 goes to the next simulation. (Intuitively speaking, this process can be regarded as a dummy-accepting path of M_1 on input z_i.)
(ii) M_2 simulates M_1 on z_i. If M_1 enters a rejecting state, then so does M_2; otherwise, it goes to the next simulation.
(4) M_2 enters an accepting state.

We can classify all possible accepting computation paths of M on input x into two groups, one of which consists of accepting paths of $M(A)$ on x (group 1), and the other consists of the remaining ones (group 2). Obviously, every accepting path in group 1 contains correct oracle answers of the oracle set A, and every accepting path in group 2 contains a wrong oracle answer. From the definition of M_2, we can easily see that every accepting path in group 1 is followed by an odd number of accepting paths in steps 3 and 4, and every accepting path in group 2 is followed by an even number of accepting paths in those steps. From this observation, it is not difficult to see that for every x, $x \in L$ if and only if the number of accepting paths of M_2 on input x is odd. We leave the verification to the interested reader. Thus L is in $\oplus P$. The other statements are immediate from the first one. □

The following theorem was shown by Schöning [15].

THEOREM 3.5 [15]. *Let* **K** *be a class of sets which is closed under* $\leq_{maj}^{P}$. *Then for all sets* $A \in \mathsf{BP} \cdot \mathbf{K}$ *and all polynomials* q, *there exist a set* B *and a polynomial* p *such that, for all* n,

$$\text{Prob}(\{y \in \{0,1\}^{p(n)} : (\forall x, |x| = n)[x \# y \in B \leftrightarrow x \in A]\}) \geqq 1 - 2^{-q(n)}.$$

LEMMA 3.6. $\oplus \cdot \mathsf{BP} \cdot \oplus P \subseteq \mathsf{BP} \cdot \oplus P$.

Proof. Let $L \in \oplus \cdot \mathsf{BP} \cdot \oplus P$. Then there exist a set $A \in \mathsf{BP} \cdot \oplus P$ and a polynomial p such that for each x, $x \in L$ if and only if

$$\|\{w : |w| = p(|x|) \text{ and } x \# w \in A\}\|$$

is odd. Furthermore, there exist a set $B \in \oplus P$, a polynomial q, and a constant $\alpha > 0$ such that for each y,

$$\text{Prob}(\{u \in \{0,1\}^{q(|y|)} : y \# u \in B \leftrightarrow y \in A\}) \geqq \tfrac{1}{2} + \alpha.$$

Since $\oplus P$ is closed under $\leq_{maj}^{P}$, we may assume, from Theorem 3.5, that for all m,

$$\text{Prob}(\{u \in \{0,1\}^{q(m)} : (\forall y, |y| = m)[y \# u \in B \leftrightarrow y \in A]\}) \geqq \tfrac{3}{4}.$$

Hence we have that for all x of length n,

$$\text{Prob}\,(\{u \in \{0,1\}^{q(n+1+p(n))}: (\forall w \in \{0,1\}^{p(n)})[x \# w \# u \in B \leftrightarrow x \# w \in A]\}) \geqq \tfrac{3}{4},$$

and hence,

$$(1) \quad \text{Prob}\,(\{u \in \{0,1\}^{q(n+1+p(n))}: \{w \in \{0,1\}^{p(n)}: x \# w \# u \in B\} = \{w \in \{0,1\}^{p(n)}: x \# w \in A\}\}) \geqq \tfrac{3}{4}.$$

For a string x of length n, assume $x \in L$. Then $\|\{w \in \{0,1\}^{p(n)}: x \# w \in A\}\|$ is odd. Hence from (1),

$$(2) \quad \text{Prob}\,(\{u \in \{0,1\}^{q(n+1+p(n))}: \|\{w \in \{0,1\}^{p(n)}: x \# w \# u \in B\}\| \text{ is odd}\}) \geqq \tfrac{3}{4}.$$

Conversely, assume $x \notin L$. Then $\|\{w \in \{0,1\}^{p(n)}: x \# w \in A\}\|$ is even. Hence from (1),

$$(3) \quad \text{Prob}\,(\{u \in \{0,1\}^{q(n+1+p(n))}: \|\{w \in \{0,1\}^{p(n)}: x \# w \# u \in B\}\| \text{ is odd}\}) \leqq \tfrac{1}{4}.$$

We now define B' and A' by

$$B' = \{x \# u \# w: |u| = q(|x|+1+p(|x|)), |w| = p(|x|), \text{ and } x \# w \# u \in B\}$$

and

$$A' = \{x \# u: |u| = q(|x|+1+p(|x|)) \text{ and } x \# u \# w \in B' \text{ for an odd number of } w \in \{0,1\}^{p(|x|)}\}.$$

Then we obtain that $\text{Prob}\,(\{u \in \{0,1\}^{q(|x|+1+p(|x|))}: x \# u \in A' \leftrightarrow x \in L\}) \geqq \frac{3}{4}$ from (2) and (3) above. It is easy to see that $B' \in \oplus\text{P}$ and $A' \in \oplus \cdot \oplus\text{P}$. Hence, A' is in $\oplus\text{P}$ from Theorem 3.4. This implies $L \in \text{BP} \cdot \oplus\text{P}$. □

LEMMA 3.7. $\text{BP} \cdot \text{BP} \cdot \oplus\text{P} = \text{BP} \cdot \oplus\text{P}$.

Proof. It suffices to show the inclusion $\text{BP} \cdot \text{BP} \cdot \oplus\text{P} \subseteq \text{BP} \cdot \oplus\text{P}$. Let $L \in \text{BP} \cdot \text{BP} \cdot \oplus\text{P}$. Then there exist a set $A \in \text{BP} \cdot \oplus\text{P}$ and a polynomial p such that for each x,

$$\text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \in A \leftrightarrow x \in L\}) \geqq \tfrac{3}{4}.$$

Furthermore, there exist a set $B \in \oplus\text{P}$ and a polynomial q such that for each y,

$$\text{Prob}\,(\{u \in \{0,1\}^{q(|y|)}: y \# u \in B \leftrightarrow y \in A\}) \geqq \tfrac{7}{8}.$$

Note that we are using Theorem 3.5 in this setting. We now define a set C by

$$C = \{x \# wu: |w| = p(|x|), |u| = q(|x \# w|) = q(|x|+1+p(|x|)), \text{ and } x \# w \# u \in B\}.$$

It is easy to see that C is in $\oplus\text{P}$. For a string x, if $x \in L$, then

$$\begin{aligned}(1) \quad & \text{Prob}\,(\{wu \in \{0,1\}^{p(|x|)+q(|x|+1+p(|x|))}: x \# wu \in C\}) \\ &= \text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \in A\}) \\ &\quad \times \text{Prob}\,(\{u \in \{0,1\}^{q(|x|+1+p(|x|))}: x \# w \# u \in B\} \mid x \# w \in A) \\ &\quad + \text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \notin A\}) \\ &\quad \times \text{Prob}\,(\{u \in \{0,1\}^{q(|x|+1+p(|x|))}: x \# w \# u \in B\} \mid x \# w \notin A) \\ &\geqq \tfrac{3}{4} \cdot \tfrac{7}{8} = \tfrac{21}{32} = \tfrac{1}{2} + \tfrac{5}{32}\end{aligned}$$

where Prob (X/Y) denotes the conditional probability of the event X under the condition Y.

Conversely, if $x \notin L$, then

$$\text{(2)} \quad \begin{aligned} &\text{Prob}\,(\{wu \in \{0,1\}^{p(|x|)+q(|x|+1+p(|x|))}: x \# wu \in C\}) \\ &= \text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \in A\}) \\ &\quad \times \text{Prob}\,(\{u \in \{0,1\}^{q(|x|+1+p(|x|))}: x \# w \# u \in B\} \mid x \# w \in A) \\ &\quad + \text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \notin A\}) \\ &\quad \times \text{Prob}\,(\{u \in \{0,1\}^{q(|x|+1+p(|x|))}: x \# w \# u \in B\} \mid x \# w \notin A) \\ &\leqq \tfrac{1}{4} \cdot 1 + 1 \cdot \tfrac{1}{8} = \tfrac{3}{8} = \tfrac{1}{2} - \tfrac{1}{8}. \end{aligned}$$

Thus we have that for every x,

$$\text{Prob}\,(\{v \in \{0,1\}^{p(|x|)+q(|x|+1+p(|x|))}: x \# v \in C \leftrightarrow x \in L\}) \geqq \tfrac{1}{2} + \tfrac{1}{8}.$$

This implies that $L \in \mathsf{BP} \cdot \oplus \mathrm{P}$. □

Now we can prove Theorem 3.1.

Proof of Theorem 3.1. We prove this theorem by induction on the levels of the polynomial-time hierarchy. The inclusion $\Sigma_0^{\mathrm{P}} = P \subseteq \mathsf{BP} \cdot \oplus \mathrm{P}$ is obvious. We now assume $\Sigma_{k-1}^{\mathrm{P}} \subseteq \mathsf{BP} \cdot \oplus \mathrm{P}$ for some $k > 0$. It is easy to see that $\mathsf{BP} \cdot \oplus \mathrm{P}$ is closed under complementation. Hence we have the inclusion $\Pi_{k-1}^{\mathrm{P}} \subseteq \mathsf{BP} \cdot \oplus \mathrm{P}$. Then,

$$\begin{aligned} \Sigma_k^{\mathrm{P}} &\subseteq \mathsf{BP} \cdot \oplus \cdot \Pi_{k-1}^{\mathrm{P}} \quad \text{(from Lemma 3.3)} \\ &\subseteq \mathsf{BP} \cdot \oplus \cdot \mathsf{BP} \cdot \oplus \mathrm{P} \quad \text{(from inductive assumption)} \\ &= \mathsf{BP} \cdot \oplus \mathrm{P} \quad \text{(from Lemma 3.6 and Lemma 3.7).} \end{aligned}$$

Thus we conclude that $\mathrm{PH} = \bigcup_{k \geqq 0} \Sigma_k^{\mathrm{P}} \subseteq \mathsf{BP} \cdot \oplus \mathrm{P}$.

It was shown by Schöning [15] that $\Pi_k^{\mathrm{P}} \subseteq \mathsf{BP} \cdot \Sigma_k^{\mathrm{P}}$ implies $\Sigma_{k+1}^{\mathrm{P}} = \Pi_{k+1}^{\mathrm{P}}$ for every $k \geqq 1$, which is regarded as a refinement of the result by Karp and Lipton [7]. Combining this result with Theorem 3.1, we observe that $\oplus \mathrm{P}$ is harder than PH unless PH collapses to a finite level. More precisely, we obtain the following corollary.

COROLLARY 3.8. *For every* $k \geqq 1$, *if* $\oplus \mathrm{P} \subseteq \Sigma_k^{\mathrm{P}}$, *then* $\mathrm{PH} = \Sigma_{k+1}^{\mathrm{P}}$. *Hence* $\oplus \mathrm{P} \subseteq \mathrm{PH}$ *implies that* PH *collapses at a finite level.*

The second statement in this corollary follows from the fact that $\oplus \mathrm{P}$ has a complete set under $\leq_{\mathrm{m}}^{\mathrm{P}}$-reducibility.

4. PP is $\leqq_{\mathrm{T}}^{\mathrm{P}}$-hard for PH. In this section, we prove the following theorem.

THEOREM 4.1. $\mathbf{C} \cdot \oplus \mathrm{P} \subseteq \mathrm{P(PP)}$.

It is easy to see that $\mathsf{BP} \cdot \oplus \mathrm{P} \subseteq \mathbf{C} \cdot \oplus \mathrm{P}$. Hence the Main Theorem in § 1 follows immediately from this theorem and from Theorem 3.1.

The following lemma plays an important role in the proof of Theorem 4.1, and depends on an interesting numerical property. For an NP-machine N and an input y, let $\# \mathrm{acc}_{\mathrm{N}}(y)$ denote the number of accepting computation paths of N on input y.

LEMMA 4.2. *Let X be a set in $\oplus \mathrm{P}$ and let q be a polynomial. Then, there exists an* NP-*machine* N_1 *such that for each input y of length n,*

(1) *if* $y \notin X$, *then* $\# \mathrm{acc}_{\mathrm{N}_1}(y) \equiv 0 \pmod{2^{q(n)}}$, *and*

(2) *if* $y \in X$, *then* $\# \mathrm{acc}_{\mathrm{N}_1}(y) \equiv -1 \pmod{2^{q(n)}}$.

Before proving this lemma, we give an intuitive explanation about our proof of Theorem 4.1 and about the role of Lemma 4.2. Let L be a set in $\mathbf{C} \cdot \oplus \mathrm{P}$. Then there exist a set $X \in \oplus \mathrm{P}$ and a polynomial p such that for every x,

$$\text{Prob}\,(\{w \in \{0,1\}^{p(|x|)}: x \# w \in X \text{ iff } x \in L\}) > \tfrac{1}{2}.$$

Let N_1 be an NP-machine satisfying the conditions in Lemma 4.2 for the set L and the polynomial $q(n)=n$. Consider an NP-machine N which operates as follows. Given an input x of length n, N first guesses a string w of length $p(n)$, and it begins to simulate N_1 on input $x\#w$. If N_1 enters an accepting state, then N enters an accepting state; otherwise, it enters a rejecting state. Let $\#X[x]$ denote the number of strings w such that $|w|=p(n)$ and $x\#w\in X$. From Lemma 4.2, it is not difficult to see that $\#\mathrm{acc}_N(x)=2^{n+1+p(n)}\cdot k_x-\#X[x]$ for some natural number $k_x\geqq 0$. By a standard binary search technique, $\#\mathrm{acc}_N(x)$ can be computed in polynomial time with an oracle set from PP. After this, we can also get the value of $\#X[x]$ within polynomial time by computing $\#\mathrm{acc}_N(x) \bmod 2^{n+1+p(n)}$, because $\#X[x]\leqq 2^{p(n)}<2^{n+1+p(n)}$. Finally, we decide to accept the input x if and only if $\#X[x]>2^{p(n)-1}$. Hence, we can conclude that $\mathbf{C}\cdot\oplus\mathrm{P}\subseteq\mathrm{P(PP)}$.

The recurrence relation in the next lemma provides the key numerical property. Intuitively speaking, it gives us a whole exponential factor of freedom in our counting.

LEMMA 4.3. *For an integer m, define the sequence* $s_0, s_1, s_2, \cdots$, *inductively by* $s_0=m$ *and for* $i\geqq 1$, $s_i=3\cdot s_{i-1}^4+4\cdot s_{i-1}^3$. *Then,*

(1) *if m is even, then for all i,* s_i *is a multiple of* 2^{2^i}, *and*

(2) *if m is odd, then for all i,* s_i+1 *is a multiple of* 2^{2^i}.

Proof. The statement (1) is obvious from the definition of the sequence. We prove (2). The case $i=0$ is obvious. We assume that for some $i>0$, $s_{i-1}=2^{2^{i-1}}\cdot k_{i-1}-1$ for some positive integer k_{i-1}. Then, from the definition of s_i,

$$\begin{aligned} s_i &= 3\cdot s_{i-1}^4+4\cdot s_{i-1}^3 \\ &= 3\cdot(2^{2^{i-1}}\cdot k_{i-1}-1)^4+4\cdot(2^{2^{i-1}}\cdot k_{i-1}-1)^3 \\ &= 3\cdot 2^{2^{i+1}}\cdot k_{i-1}^4-8\cdot 2^{3\cdot 2^{i-1}}\cdot k_{i-1}^3+6\cdot 2^{2^i}\cdot k_{i-1}^2-1 \\ &= 2^{2^i}\cdot(3\cdot 2^{2^i}\cdot k_{i-1}^4-8\cdot 2^{2^{i-1}}\cdot k_{i-1}^3+6\cdot k_{i-1}^2)-1. \end{aligned}$$

Hence s_i is a multiple of 2^{2^i}. □

Accordingly, we make the following recursive definition of a function $f_N:\Sigma^*\times\mathbf{N}\to\mathbf{N}$, where Σ is the input alphabet of a given NTM N.

DEFINITION 4.4. For an NP-machine N and an input y, define

$$f_N(y,0)=\#\mathrm{acc}_N(y),\quad \text{and for } i\geqq 1,$$

$$f_N(y,i)=3\cdot(f_N(y,i-1))^4+4\cdot(f_N(y,i-1))^3.$$

LEMMA 4.5. *Let* N *be an* NP-*machine, and let* $q(n)$ *be a polynomial. Then for all input y of length n,*

(1) *if* $\#\mathrm{acc}_N(y)$ *is even, then* $f_N(y,\lceil\log_2 q(n)\rceil)\equiv 0 \pmod{2^{q(n)}}$, *and*

(2) *if* $\#\mathrm{acc}_N(y)$ *is odd, then* $f_N(y,\lceil\log_2 q(n)\rceil)\equiv -1 \pmod{2^{q(n)}}$.

Proof. Since $2^{2^{\lceil\log_2 q(n)\rceil}}=2^{q(n)+k}=2^{q(n)}\cdot 2^k$ for some natural number k, this follows immediately from Lemma 4.3. □

Given a set $X\in\oplus\mathrm{P}$, it follows from the definition of $\oplus\mathrm{P}$ that there is an NP-machine N such that for all y, $y\in X$ if and only if $\#\mathrm{acc}_N(y)$ is odd. Hence, to obtain Lemma 4.2, it remains to show how to construct an NP-machine Q such that for all y,

$$\#\mathrm{acc}_Q(y)=f_N(y,\lceil\log_2 q(n)\rceil).$$

This is accomplished in the following lemma.

LEMMA 4.6. *Let* N *be an* NP-*machine, and let t be a polynomial which bounds the runtime of* N. *Then we can find an* NTM Q *which takes inputs of the form* $y \# 1^i$, *and a constant* $c>0$ *such that for all* y, i:

(1) $\#\mathrm{acc}_{\mathrm{Q}}(y \# 1^i)=f_{\mathrm{N}}(y, i)$, *and*

(2) *all computations of* Q *on input* $y \# 1^i$ *halt within* $c \cdot 4^i \cdot (t(|y|)+1)$ *steps.*

Proof. Intuitively speaking, the required machine Q is designed so that for each input $y \# 1^i$, it executes itself recursively on input $y \# 1^{i-1}$ according to the definition of f_{N}. This can be done by using stack operations. Furthermore, since the depth of recursive executions for input $y \# 1^i$ is at most i, and at most four sequential calls are made at each level, the required time bound is obtained. We now describe Q as a recursive procedure, using a stack for the recursive executions.

```
PROCEDURE Q(y, i), where the input is written in the form y # 1^i.
Step 1: if i = 0 then simulate M on input y;
                    if M enters an accepting state
                      then return "ACCEPT" else return "REJECT";
Step 2: guess one of the following subprocesses nondeterministically;
        (subprocess 1)
        branch away nondeterministically into three branches;
        execute the following in each branch;
        for j := 1 to 4 do
             execute Q(y, i-1) recursively;
             {this can be done by pushing i, j and return position into a stack before
              execution and by popping those off the stack after execution}
             if this call to Q(y, i-1) returns "REJECT" then return "REJECT"
        od;
        return "ACCEPT";
        (subprocess 2)
        branch away nondeterministically into four branches;
        execute the following in each branch;
        for j := 1 to 3 do
             execute Q(y, i-1) recursively;
             {this can be done by pushing i, j and return position into a stack before
              execution and by popping those off the stack after execution}
             if this call to Q(y, i-1) returns "REJECT" then return "REJECT"
        od;
        return "ACCEPT."
```

By induction on i, it is not difficult to show that for each input $y \# 1^i$, the number of accepting computation paths of Q is equal to $f_{\mathrm{N}}(y, i)$. The essence of this proof is to estimate the runtime of the above machine. Let $y \# 1^i$ be an input for Q and let $T(y, i)$ denote the runtime of Q on input $y \# 1^i$. It is not difficult to see that stack operations and the other bookkeeping operations in Step 2 can be done within time at most $O(i)$, say $c \cdot i+c$ for some $c>0$, if we denote natural numbers by unary notation. Furthermore, the operations in Step 1 can be done within a constant time, say $c>0$. Then, we obtain the following inequalities from the definition of M_1:

$$T(y, 0) \leqq t(|y|)+c, \tag{1}$$

and

$$T(y, i) \leqq 4 \cdot (T(y, i-1)+c \cdot i+c) \quad \text{for each } i>0. \tag{2}$$

From this, we have:

(3) $T(y, i) \leqq 4^i \cdot t(|y|) + \Sigma_{k=1}^{i} 4^k \cdot (c \cdot i + c) = 4^i \cdot t(|y|) + \frac{4}{3} \cdot (4^i - 1) \cdot (c \cdot i + c)$.

Thus we finally have $T(y, i) \leqq O(4^i \cdot (t(|y|) + i))$. This completes the proof.

Proof of Lemma 4.2. Let N_1 on input y simulate Q of Lemma 4.6 on input $y \# 1^{\lceil \log_2 q(|y|) \rceil}$. Then N_1 runs in polynomial time in $|y|$, and satisfies the properties required in Lemma 4.2. □

Before deducing Theorem 4.1, we state and prove a technically stronger result. A function $h: \Sigma^* \to \mathbf{N}$ is said to belong to the class $\#\mathrm{P}$ [23], [24] if there is an NP-machine N such that for all $x \in \Sigma^*$, $h(x) = \#\mathrm{acc}_N(x)$. Then $\mathrm{P}^{\#\mathrm{P}[1]}$ stands for the class of sets which can be solved in polynomial time with one free evaluation of a $\#\mathrm{P}$ function. Papadimitriou and Zachos [13] showed that $\mathrm{P}^{\mathrm{NP}[\log]} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$ (calling the latter class "$\#\mathrm{P}$"). We show that the whole polynomial-time hierarchy is contained in $\mathrm{P}^{\#\mathrm{P}[1]}$, as a consequence of Theorem 4.7.

THEOREM 4.7. $\mathbf{C} \cdot \oplus \mathrm{P} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

Proof. Let $L \in \mathbf{C} \cdot \oplus \mathrm{P}$. Then there is a set $X \in \oplus \mathrm{P}$ and a polynomial p such that for all x, putting $W_x = \{w \in \{0,1\}^{p(|x|)}: x \# w \in X\}$, we have $x \in L$ if and only if $\| W_x \| > 2^{p(|x|)-1}$. Then from Lemma 4.2, we can find an NP-machine N such that for all inputs y, with $m = |y|$,

(1) if $y \in X$, then for some integer $\alpha_y > 0$, $\#\mathrm{acc}_N(y) = 2^m \cdot \alpha_y - 1$, and

(2) if $y \notin X$, then for some integer $\beta_y \geqq 0$, $\#\mathrm{acc}_N(y) = 2^m \cdot \beta_y$.

Now let Z be an NTM which on every input x of length n does this:

(1) Guess $w \in \{0,1\}^{p(n)}$.

(2) Simulate N on input $x \# w$, accepting if and only if N accepts.

Then Z clearly runs in polynomial time. Now writing $\tilde{W}_x$ for $\{0,1\}^{p(n)} - W_x$, we have:

$$\begin{aligned}\#\mathrm{acc}_Z(x) &= \sum_{w \in W_x} \#\mathrm{acc}_N(x \# w) + \sum_{w \in \tilde{W}_x} \#\mathrm{acc}_N(x \# w) \\ &= \sum_{w \in W_x} (2^{|x \# w|} \cdot \alpha_{x \# w} - 1) + \sum_{w \in \tilde{W}_x} (2^{|x \# w|} \cdot \beta_{x \# w}) \\ &= 2^{|x|+1+p(|x|)} \cdot \left(\sum_{w \in W_x} \alpha_{x \# w} + \sum_{w \in \tilde{W}_x} \beta_{x \# w} \right) - \| W_x \|.\end{aligned}$$

Since every $\alpha_{x \# w}$ and $\beta_{x \# w}$ is integral, it follows that $\#\mathrm{acc}_Z(x) + \| W_x \|$ is a multiple of $2^{|x|+1+p(|x|)}$. Since $\| W_x \| \leqq 2^{p(|x|)} < 2^{|x|+1+p(|x|)}$, it follows that $\| W_x \|$ can be computed simply by complementing the last $p(|x|)$ bits of $\#\mathrm{acc}_Z(x)$ in binary notation. That is to say, $x \in L$ if and only if the $p(|x|)$th bit of $\#\mathrm{acc}_Z(x)$ from the right is a "0." Since Z is a polynomial-time-bounded NTM, $\#\mathrm{acc}_Z(\cdot)$ is a $\#\mathrm{P}$ function, and the theorem follows. □

Proof of Theorem 4.1. It is well known that for every $\#\mathrm{P}$ function h, its *graph* $\{x \# k: h(x) \leqq k\}$ belongs to PP [14]. By the standard binary search technique, $h(x)$ can be computed with $O(|x|)$-many queries to its graph. So $\mathbf{C} \cdot \oplus \mathrm{P}$, and hence PH, is included in P(PP). □

The following corollary is straightforward from the Main Theorem.

COROLLARY 4.8. *For each* $k \geqq 0$, *if* $\mathrm{PP} \subseteq \Sigma_k^{\mathrm{P}}$, *then* PH *collapses to* Σ_k^{P}. *Furthermore, if* $\mathrm{PP} \subseteq \mathrm{PH}$, *then* PH *collapses to a finite level.*

Proof. Assume $\mathrm{PP} \subseteq \Sigma_k^{\mathrm{P}}$. It is well known that PP is closed under complementation. Hence we have $\mathrm{PP} \subseteq \Sigma_k^{\mathrm{P}} \cap \Pi_k^{\mathrm{P}}$ from the assumption. It is also well known that $\mathrm{P}(\Sigma_k^{\mathrm{P}} \cap \Pi_k^{\mathrm{P}}) \subseteq \Sigma_k^{\mathrm{P}}$ for each $k \geqq 0$. From the main theorem, we have

$$\mathrm{PH} \subseteq \mathrm{P}(\mathrm{PP}) \subseteq \mathrm{P}(\Sigma_k^{\mathrm{P}} \cap \Pi_k^{\mathrm{P}}) \subseteq \Sigma_k^{\mathrm{P}}.$$

The second statement is easily obtained from the fact that PP has a complete set under $\leq_m^P$-reducibility. □

At the end of this section, we observe a result stronger than the Main Theorem. It was shown by Köbler et al. [10] that PP(BPP) = PP. Furthermore, the equality can be relativized to all oracle sets. More precisely, we have the following result.

THEOREM 4.9 [10]. *For all oracle sets A*, $PP(BPP(A)) = PP(A)$.

From this theorem, we have the following theorem.

THEOREM 4.10. PP(PH) *is included in* P(PP).

Proof. It is easy to see that $PP(PH) \subseteq PP(\mathsf{BP} \cdot \oplus P) \subseteq PP(BPP(\oplus P))$. These inclusions follow from Theorem 3.1 and from the definition of BP-operator. From Theorem 4.9, we have $PP(PH) \subseteq PP(\oplus P)$. It is not hard to show that $PP(\oplus P) = \mathbf{C} \cdot P(\oplus P) = \mathbf{C} \cdot \oplus P$. Some techniques for showing this have appeared in [21], [26]. Hence we obtain this theorem from Theorem 4.1.

5. Concluding remarks. In this paper, we showed that every set in PH (and in PP(PH)) is polynomial-time Turing reducible to a set in PP. We also show a similar result about $\oplus P$. There are some further questions that are related to this work. A simple question is whether we can show, by using a different kind of reducibility such as polynomial-time truth-table reducibility, that PH is reducible to PP. In fact, we showed that PH is included in $P^{\#P[1]}$; this is a somewhat stronger statement than $PH \subseteq P(PP)$. On the other hand, it is well known that every set which is $\leq_{tt}^P$-reducible to a set in PP is in $P^{\#P[1]}$; but the converse is unknown. Hence the answer to the above question will give us a somewhat stronger result than the present result. The other interesting question is whether $C_=P$ [26], [21] is as hard as PH. A more important question is whether NP(PP) is included in P(PP), or whether PP(PP) is included in P(PP). It is also interesting to find oracle sets that separate those classes from each other.

Acknowledgment. I am very thankful to Osamu Watanabe for helpful discussions and some nice advice, to Lane Hemachandra for his comments on this work, and to Kenneth Regan for his many suggestions on the earlier version of this paper. I am very thankful to the Program Committee of the 4th IEEE Conference on Structure in Complexity Theory in 1989 for giving me an opportunity to talk about this result at the conference; concerning this, I have to thank Richard Beigel, Lane Hemachandra, and Gerd Wechsung for giving much of their lecture time to me. I want to thank the referees of this paper. In particular, one of the referees gave me many suggestions which made the quality of this paper much better. I would also like to thank Ron Book and Janos Simon for their suggestions.

REFERENCES

[1] D. ANGLUIN, *On counting problems and the polynomial-time hierarchy*, Theoret. Comput. Sci., 12 (1980), pp. 161–173.

[2] J. L. BALCÁZAR, R. V. BOOK, AND U. SCHÖNING, *The polynomial-time hierarchy and sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 603–617.

[3] R. BEIGEL, L. A. HEMACHANDRA, AND G. WECHSUNG, *On the power of probabilistic polynomial-time*: $P^{NP[\log]} \subseteq PP$, in Proc. 4th IEEE Conference on Structure in Complexity Theory, 1989, pp. 225–227.

[4] JIN-YI CAI AND L. A. HEMACHANDRA, *On the power of parity polynomial time*, in Proc. Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 349, 1989, Springer-Verlag, Berlin, pp. 229–240.

[5] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.

[6] L. A. HEMACHANDRA, *On ranking*, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 103-117.

[7] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 302-309.

[8] K. KO, *Some observations on the probabilistic algorithms and* NP-*hard problems*, Inform. Process. Lett., 14 (1982), pp. 39-43.

[9] J. KÖBLER, U. SCHÖNING, AND J. TORAN, *On counting and approximation*, Acta Inform., 26 (1989), pp. 363-379.

[10] J. KÖBLER, U. SCHÖNING, S. TODA, AND J. TORAN, *Turing machines with few accepting computations and low sets for* PP, in Proc. 4th IEEE Conference on Structure in Complexity Theory, 1989, pp. 208-215.

[11] C. LAUTEMAN, BPP *and the polynomial-time hierarchy*, Inform. Process. Lett., 14 (1983), pp. 215-217.

[12] T. J. LONG AND A. L. SELMAN, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618-627.

[13] C. H. PAPADIMITRIOU AND S. ZACHOS, *Two remarks on the power of counting*, in Proc. 6th Gesellschaft für Informatik Conference on Theoretical Computer Science, Lecture Notes in Computer Science 145, 1983, Springer-Verlag, Berlin, pp. 269-276.

[14] J. SIMON, *On the difference between one and many*, in Proc. 4th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 52, 1977, Springer-Verlag, Berlin, pp. 480-491.

[15] U. SCHÖNING, *Probabilistic complexity classes and lowness*, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 2-8; also in J. Comput. System Sci., 39 (1989), pp. 84-100.

[16] ———, *The power of counting*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 2-9.

[17] M. SIPSER, *A complexity theoretic approach to randomness*, in Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 330-335.

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

[19] ———, *On approximation algorithms for* #P, SIAM J. Comput., 14 (1985), pp. 849-861.

[20] S. TODA, *Restricted relativizations of probabilistic polynomial-time*, Theoret. Comput. Sci., 1990, accepted.

[21] J. TORAN, *An oracle characterization of the counting hierarchy*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 213-223.

[22] L. G. VALIANT, *Relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20-23.

[23] ———, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.

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

[25] L. G. VALIANT AND V. V. VAZIRANI, NP *is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85-93.

[26] K. WAGNER, *The complexity of combinatorial problems with succinct input representation*, Acta Inform., 23 (1986), pp. 325-356.

[27] ———, *Some observations on the connection between counting and recursion*, Theoret. Comput. Sci., 47 (1986), pp. 131-147.

[28] C. WRATHALL, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23-33.

[29] S. ZACHOS, *Robustness of probabilistic computational complexity classes under definitional perturbations*, Inform. and Control, 54 (1982), pp. 143-154.

[30] S. ZACHOS AND H. HELLER, *A decisive characterization of* BPP, Inform. and Control, 69 (1986), pp. 125-135.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 878-887, October 1991
© 1991 Society for Industrial and Applied Mathematics
005

SELECTION NETWORKS*

NICHOLAS PIPPENGER†

Abstract. An upper bound asymptotic to $2n \log_2 n$ is established for the number of comparators required in a network that classifies n values into two classes, each containing $n/2$ values, with each value in one class less than or equal to each value in the other. (The best lower bound known for this problem is asymptotic to $(n/2) \log_2 n$.)

Key words. comparator, classifier, expanding graph, random walk

AMS(MOS) subject classifications. 68E05, 94C10

1. Introduction. The selection networks of which we speak in this paper are comparator networks (see Knuth [K]) that classify a set of n values into two classes, with each of the values in one class being at least as large as all of those in the other. In this paper we shall confine our attention to the simplest case, in which n is even and the two classes each contain $n/2$ values, but similar methods apply to classes of unequal cardinality, as well as to the problem of selecting the value having a prescribed rank, such as the median.

We shall present an upper bound asymptotic to $2n \log_2 n$ for the number of comparators needed to construct such a network. Alekseev [A1] has given a lower bound asymptotic to $(n/2) \log_2 n$. Some perspective on the gap between these bounds is gained by considering the analogous problem of determining the median of n values with an adaptive sequence of comparisons: here the best upper bound known is asymptotic to $3n$ (see Schönhage, Paterson, and Pippenger [S]), and the best lower bound known is asymptotic to $2n$ (see Bent and John [Be]).

The classifying problem has traditionally been considered in connection with the problem of sorting n values into order. In 1983, Ajtai, Komlós, and Szemerédi [Aj] showed that $O(n \log n)$ comparators are sufficient for sorting, and this bound obviously applies to classifying as well. The constant factor implicit in their original proof is enormous, however, and further efforts to refine their ideas have not brought it below 1000 (see Paterson [Pa]). Our classifiers are based on the same fundamental idea as their sorters; our only contribution is to show that in the context of classifiers, it yields both a much simpler proof and a much smaller constant.

Though we shall confine ourselves to proving the result stated above, two additional points should be mentioned. First, we prove the existence of classifying networks without giving an explicit construction. This situation arises from the use of expanding graphs; by exploiting known explicit constructions for expanding graphs (see Pippenger [Pi, § 3.2]), and by accepting a somewhat larger bound (the best we have been able to obtain is slightly less than $6n \log_2 n$), we could give a completely explicit construction. Second, the networks we describe have depth $\Omega((\log n)^2)$; with more care in the construction and proof, we could establish a bound of $O(\log n)$. Our method does not seem well suited to optimizing the depth, however, and we have not made any attempt to obtain the sharpest possible result in this direction.

* Received by the editors April 30, 1990; accepted for publication December 5, 1990. This research was partially supported by Natural Sciences and Engineering Research Council of Canada operating grant and a British Columbia Advanced Systems Institute fellowship award.

† Department of Computer Science, University of British Columbia, Vancouver, British Columbia V6T 1W5, Canada.

2. Expanding graphs. We shall need some results concerning expanding graphs; these will be obtained as special cases of a general result due to Bassalygo [B], to whom we refer for the proof.

A bipartite graph with n "left" vertices and m "right" vertices will be called an *(α, β)-expander* if any k of its left vertices ($k \leqq \lfloor \alpha n \rfloor$) are connected to at least $\lfloor \beta k \rfloor$ right vertices (the set A of left vertices is connected to the set B of right vertices if at least one edge from A leads to each right vertex b, $b \in B$; $\lfloor x \rfloor$ is the integer part of x).

LEMMA 2.1 (Bassalygo). *For any positive integers q and p, any reals α and β ($0 < \alpha < p/\beta q < 1$), and any sufficiently large n ($n \geqq n_0(\alpha, \beta, q, p)$), there exists an (α, β)-expander with qn left vertices and pn right vertices, for which the number of edges does not exceed $spqn$, where s is any integer greater than*

$$\frac{H(\alpha)+(p/q)H(\alpha\beta q/p)}{pH(\alpha)-\alpha\beta q H(p/\beta q)}; \qquad H(x) = -x \log x - (1-x)\log(1-x), \qquad 0<x<1.$$

The proof of Lemma 2.1 considers only graphs in which every left vertex meets sp edges and every right vertex meets sq edges. This observation will be important when we consider the depth, rather than merely the size, of networks.

We shall use this lemma with $p = q = 1$. We shall let α and β depend on a new parameter ϑ by $\alpha = \vartheta$ and $\beta = (1-\vartheta)/\vartheta$. For every $\vartheta > 0$, there is a value of s that satisfies the hypothesis of Lemma 2.1.

The proof of Lemma 2.1 may be regarded as considering a probability distribution over graphs, and when $p = q$, this distribution is invariant under the exchange of left and right vertices. The proof also shows, not merely that there exists a graph with the prescribed expansion property, but that almost all considered graphs have this property. In particular, a majority of the considered graphs have this property. It follows that there exists a graph such that both it and the graph obtained from it by exchanging left and right vertices have the prescribed expansion property.

Combining these elaborations of Lemma 2.1, we obtain the following corollary.

COROLLARY 2.2. *For every $\vartheta > 0$, there exists an s such that for all sufficiently large n (depending on ϑ) there exists a bipartite graph with n left vertices, n right vertices, and s edges meeting each vertex such that* (1) *every set of $k \leqq \vartheta n$ left vertices is connected to at least $(1-\vartheta)k/\vartheta$ right vertices, and* (2) *every set of $k \leqq \vartheta n$ right vertices is connected to at least $(1-\vartheta)k/\vartheta$ left vertices.*

Returning to Lemma 2.1 with $p = q = 1$, if we take $s = 4$ and choose $\beta < 3$, then the hypothesis is satisfied for all sufficiently small $\alpha > 0$ (depending on β). Thus we also obtain the following corollary.

COROLLARY 2.3. *For every $\beta < 3$, there exists an $\alpha > 0$ such that for all sufficiently large n (depending on β), there exists a bipartite graph with n left vertices, n right vertices, and four edges meeting every vertex such that* (1) *every set of $k \leqq \alpha n$ left vertices is connected to at least βk right vertices, and* (2) *every set of $k \leqq \alpha n$ right vertices is connected to at least βk left vertices.*

3. Classifiers. A comparator network with $2m$ inputs and two sets of m outputs, the "left" outputs and the "right" outputs, is an *α-weak approximate classifier with tolerance ϑ* (or simply an *α-weak ϑ-classifier*) if, for any assignment of values to the inputs and positive integer $k \leqq \alpha m$, (1) at most ϑk of the k smallest values appear at right outputs, and (2) at most ϑk of the k largest values appear at left outputs. A 1-weak approximate classifier with tolerance ϑ will be called an *approximate classifier with tolerance ϑ* (or simply a *ϑ-classifier*). An approximate classifier with tolerance 0 will be called a *classifier.*

Classifiers are the goal of our construction. Approximate classifiers and weak approximate classifiers are the ultimate building blocks of our construction. These building blocks are secured by a lemma that is a slight generalization of the most basic lemma of Ajtai, Komlós, and Szemerédi [Aj], from which the proof is easily adapted (see also Pippenger [Pi, § 3.2]).

Let G be a bipartite graph with n left vertices and n right vertices, in which every vertex meets s edges. The edges of G may be decomposed into s perfect matchings $E_1, \cdots, E_s$ between the left and right vertices. We may regard each perfect matching E_r as a comparator network, by taking a comparator for each edge in E_r, labelling the inputs of the comparator with the vertices met by the edge, labelling the smaller output of the comparator with the left vertex met by the edge, and labelling the larger output of the comparator with the right vertex met by the edge. We may then combine the comparator networks $E_1, \cdots, E_s$ into a single comparator network by identifying the outputs of E_r with the corresponding inputs of E_{r+1} for each $1 \leqq r \leqq s-1$. We shall denote the resulting comparator network by G; this notation is ambiguous, since different decompositions of the bipartite graph yield different comparator networks, but this ambiguity will not be important to us.

LEMMA 3.1 (Ajtai, Komlós, and Szemerédi). *Let G be a bipartite graph with n left vertices and n right vertices and s edges meeting every vertex in which* (1) *any set of $k \leqq \alpha\vartheta n$ left vertices is connected to at least $(1-\vartheta)k/\vartheta$ right vertices, and* (2) *any set of $k \leqq \alpha\vartheta n$ right vertices is connected to at least $(1-\vartheta)k/\vartheta$ left vertices. Let the left and right outputs of the comparator network G be those labelled by the left and right vertices, respectively. Then G is an α-weak ϑ-classifier.*

4. Recursive construction. A comparator network with $2m$ inputs, l outputs labelled as "low," l outputs labelled as "high," and $2m-2l$ outputs labelled as "middle" is a *strong partial classifier* if, for any assignment of values to the inputs, only values among the $m/2$ smallest appear at low outputs and only values among the $m/2$ largest appear at high outputs. A strong partial classifier is less than a classifier in that there are some outputs, the middle outputs, at which any value may appear; but it is more than a classifier in that fewer values can appear at the low and high outputs. Our goal in this section is to show how strong partial classifiers can be assembled to form a classifier.

It will be convenient to use strong partial classifiers for which the number of inputs is of the form 2^ν or $3 \cdot 2^\nu$, with ν a positive integer; numbers of this form will be called *magic.* If n is any even positive integer, the largest magic number not exceeding n will be called the *magic part* of n; it is even and at least $2n/3$.

Consider the following recursive construction for a classifier with n inputs. Let $2m$ denote the magic part of n. Feed $2m$ of the inputs into a strong partial classifier with $2m$ inputs. Feed the remaining $n-2m$ inputs, together with the $2m-2l$ middle outputs of the strong partial classifier into a classifier with $2n-2l$ inputs. The left and right outputs of the combined network will be the low and high outputs, respectively, of the strong partial classifier, together with the left and right outputs, respectively, of the constituent classifier.

A value appearing at a low output of the strong partial classifier must be among the $m/2$ smallest of the $2m$ values at its inputs, and thus among the $(m/2)+(n-2m) = n-3m/2$ smallest of all n values. Since $2m \geqq 2n/3$, it must be among the $n/2$ smallest of all n values. Similarly, a value appearing at a high ouput of the strong partial classifier must be among the $n/2$ largest of all n values. Thus, of the $n/2$ largest and $n/2$ smallest values, equal numbers appear at the inputs of the classifier with $n-2l$ inputs. It follows that any value appearing at a left output of this classifier must be

among the $n/2$ smallest, and any value appearing at a right output must be among the $n/2$ largest. Thus the combined network is indeed a classifier.

LEMMA 4.1. *Let $C>0$ be a constant. Suppose that, for every $\varepsilon>0$ and all sufficiently large m (depending on ε), there exists a strong partial classifier with $2m$ inputs, l low outputs, and l high outputs, and size at most $(C+\varepsilon)l\log_2 m$. Then for every $\varepsilon>0$ and all sufficiently large n (depending on ε), there exists a classifier with n inputs and size at most $(C/2+\varepsilon)n\log_2 n$.*

Proof. Apply the recursive construction until the number of inputs of the strong partial classifier that is needed is too small for the hypothesis to apply. Terminate the recursion with a sorting network using $\binom{m}{2}$ comparators. The size of this final sorting network depends only on ε, and thus is at most $(\varepsilon/2)n\log_2 n$ for all sufficiently large n (depending on ε).

Let $2m_1,\cdots,2m_s$ denote the numbers of inputs of the strong partial classifiers, and let $l_1,\cdots,l_r$ denote the numbers of low outputs. We have $2m_r\leqq n$ for all $1\leqq r\leqq s$ and, since each left output of the combined network is a low output of at most one strong partial classifier, $\sum_{1\leqq r\leqq s} l_r\leqq n/2$. Thus the total size of all the strong partial classifiers is at most $\sum_{1\leqq r\leqq s}(C+\varepsilon)l_r\log_2 m_r\leqq(C/2+\varepsilon/2)n\log_2 n$. Adding the bound $(\varepsilon/2)n\log_2 n$ for the final sorter yields $(C/2+\varepsilon)n\log_2 n$. □

5. Crude classification trees. This section introduces classification trees, the basic tool we shall use to construct strong partial classifiers. We shall begin with a crude version of the construction, and later refine it to obtain our final bound.

Set $\vartheta=\frac{1}{8}$. Corollary 2.2 and Lemma 3.1 then yield constants s_0 and n_0 such that for all $n\geqq n_0$, there is a ϑ-classifier with $2n$ inputs and depth s_0. (A simple calculation shows that $s_0=28$. The determination of n_0 would require scrutiny of the proof of Lemma 2.1, but this proof consists of explicit estimates, so that n_0 is at least effectively calculable. The actual values of these constants will not be important to us.)

Suppose that we wish to construct a strong partial classifier with $2m$ inputs, where $2m$ is a magic number. Feed the $2m$ inputs into a ϑ-classifier with $2m$ inputs. This approximate classifier has m left outputs and m right outputs. Feed each of these sets of outputs into a ϑ-classifier with m inputs. These two approximate classifiers have four sets of outputs. Feed each of these sets into a ϑ-classifier with $m/2$ inputs, and continue in this way until the sets of outputs of the approximate classifiers have cardinality less than $2n_0$. The result is a tree of approximate classifiers that we shall call a *classification tree.* At its root are $2m$ inputs, and at its leaves are sets of outputs each containing fewer than $2n_0$ outputs.

The next step will be to label the outputs as low, high, and middle in such a way that the result is a strong partial classifier. When, as we do this, we assign the same label to all the outputs in a subtree, we may prune away that subtree, and affix the label to the outputs of the approximate classifier feeding the subtree. A large fraction of the tree will be eliminated in this way.

We begin by labelling as middle the right outputs of the left child of the root, and the left outputs of the right child of the root (and pruning away the subtrees below). We shall label as low some of the outputs in the subtree fed by the left outputs of the left child, and as high some of the outputs in the subtree fed by the right outputs of the right child. We shall now describe which outputs are to be labelled as low. The mirror image of this procedure will label an equal number of outputs as high.

Consider the $m/2$ smallest values assigned to the inputs, since it is these that are eligible to appear at an output labelled as low. We shall call these $m/2$ values *good*, and the other $3m/2$ values *bad.*

At most, a fraction $\vartheta = \frac{1}{8}$ of the good values can appear at right outputs of the approximate classifier at the root, and at most a fraction $\frac{1}{8}$ can appear at right outputs of the approximate classifier that is its left child. Thus at least a fraction $1 - (\frac{1}{8}) - (\frac{1}{8}) = \frac{3}{4}$ of the good values appear at left outputs of the left child. Since the number of good values equals the number of left outputs of the left child, at most a fraction $1 - (\frac{3}{4}) = \frac{1}{4}$ of the values appearing at these outputs are bad.

We may characterize the set of left outputs of the left child by its cardinality $m/2$ and its "impurity" $\frac{1}{4}$ (the largest possible fraction of its values that could be bad). Suppose now that we have a set of outputs of some approximate classifier with cardinality k and impurity η. First, if $\eta > \frac{1}{2}$, we shall label these outputs as middle (and prune away the subtree below). Second, if $\eta k < 1$, then not a single bad value can appear at one of these outputs; thus we shall label them as low (and prune away the subtree below). Finally, if $\eta \leqq \frac{1}{2}$ and $\eta k \geqq 1$, then we shall consider the sets of outputs of the child. The set of left outputs has cardinality $k/2$ and (by Lemma 3.1) impurity $2\vartheta\eta = \eta/4$, and the set of right outputs has cardinality $k/2$ and impurity 2η (the factors of 2 in the impurities arise because we are considering a fraction of half as many things). We may continue in this way along each path in the tree until we assign a label or reach a leaf. If we reach a leaf, we shall label its outputs as middle if $\eta k \geqq 1$, and as low if $\eta k < 1$.

The first question we shall ask is: what fraction of the outputs are labelled as middle by being in a set with impurity exceeding $\frac{1}{2}$? To answer this question, we shall consider the following random walk on the integers. Start at the position 2, $Z_0 = 2$. At each step, independently move to the position one smaller, $Z_{t+1} = Z_t - 1$, or two larger, $Z_{t+1} = Z_t + 2$, with equal probabilities. What is the probability of ever reaching a position smaller than 1? Since the walk is confined to the integers, this is the probability of ever reaching the position 0, $Z_t = 0$. The answer to this question is an upper bound to the fraction of the outputs that are labelled as middle by being in a set with impurity exceeding $\frac{1}{2}$, as can be seen by considering the correspondence between paths in the tree and walks, where the number of levels from the root corresponds to time in the walk, and the negative of the logarithm (to base 2) of the impurity corresponds to position in the walk.

In the present instance, the probability of ever reaching the position 0 can be determined explicitly and is $(3-\sqrt{5})/2 = 0.382 \cdots$. To see this, let $f(x)$ denote the power series in x in which the coefficient of x^t is the number of walks that start at 1 and reach 0 for the first time at time t. Then $f(x)^2$ is the power series for walks that start at 2 and reach 0 for the first time at time t, since each such walk can be uniquely parsed into two subwalks according to the time at which it first reaches 1, and the numbers of possibilities for both subwalks are counted by $f(x)$. Thus the probability we seek is $f(\frac{1}{2})^2$, so it will suffice to show that $f(\frac{1}{2}) = (-1+\sqrt{5})/2 = 0.618 \cdots$. Let $g(x)$ count the number of walks that start at 1 and return to 1 for the first time at time t. Then $g(x) = xf(x)^2$, since such a walk must go to 3 on the first step, then return to 1 for the first time in $t-1$ more steps. On the other hand, $f(x) = x + xg(x) + xg(x)^2 + \cdots = x/(1-g(x))$, since the walks counted by $f(x)$ may be classified according to the number of times they visit 1 before reaching 0. Thus $f(x)$ satisfies the equation $xf(x)^3 - f(x) + x = 0$, so that $f(\frac{1}{2})$ satisfies $f(\frac{1}{2})^3 - 2f(\frac{1}{2}) + 1 = 0$, which yields the stated result.

Next we shall ask: what fraction of the outputs are labelled as middle by being in a leaf that is not pruned away? Such a leaf has cardinality at most $2n_0$, and thus it must have impurity at least $\frac{1}{2}n_0$ to avoid being labelled as low. Let d denote the number of levels of approximate classifiers in the tree. Rephrased in terms of random walks, our question becomes: what is the probability of being at a position at most $c_0 = \log_2 n_0$

at time $d-2$? (The first two levels of the tree do not correspond to steps of the random walk.) We shall answer this question with a lemma that goes beyond our present needs, but which will be applied repeatedly later.

We shall consider random walks with discrete time indexed by the natural numbers and discrete positions indexed by the integers. We shall assume that the steps are independent and identically distributed, but we shall allow the steps to have any probability distribution on a finite set of integers. We shall say that such a random walk is *positively biased* if the expectation of a step is positive. (In the present instance, the step is uniformly distributed on the set $\{-1, 2\}$, and the expectation is $(-1+2)/2 = \frac{1}{2} > 0$.)

LEMMA 5.1. *Let Z_t, $t = 0, 1, 2, \cdots$, be a positively biased random walk starting at 0, and let c be any position. Then there exist constants A and $b < 1$ such that for all t, the probability that Z_t is at most c does not exceed Ab^t.*

Proof. Let $\Phi(\xi) = \mathrm{Ex}(\exp -(\xi Z_1))$. The power series expansion of $\Phi(\xi)$ is $1 - \xi\, \mathrm{Ex}\,(Z_1) + O(\xi^2)$. Since $\mathrm{Ex}\,(Z_1) > 0$, we can choose $\xi_0 > 0$ sufficiently small so that $\Phi(\xi_0) < 1$. Since the steps are independent and identically distributed, we have $\mathrm{Ex}\,(\exp - (\xi_0 Z_t)) = \Phi(\xi_0)^t$. If $Z_t \leqq c$, then $\exp - (\xi_0 Z_t) \geqq \exp - (\xi_0 c)$. Thus, by Markov's inequality, the probability that $Z_t \leqq c$ is at most $\Phi(\xi_0)^t / \exp - (\xi_0 c)$, so we may take $A = \exp\,(\xi_0 c)$ and $b = \Phi(\xi_0)$. □

We may now apply Lemma 5.1 with $t = d - 2 \geqq \log_2 (m/4n_0)$ and conclude that the fraction of outputs that are labelled as middle by being in a leaf that is not pruned away is at most $Ab^d \leqq Cm^{-e}$, where C and $e > 0$ are constants. The only feature of this bound that is relevant to our present purposes is that it tends to zero, even when multiplied by $d \leqq \log_2 m$.

Finally, we shall ask: what is the size of the approximate classifier constructed in this way? To answer this question, we shall again transform it into a question about random walks. In a "synchronous" comparator network (in which the two inputs of any comparator are at the same depth), each comparator contributes 2 to the sum of the depths of the outputs. Thus the number of comparators is $n/2$ times the average depth of the outputs. Since the depth of each approximate classifier is s_0, the number of comparators is $s_0 n/2$ times the average level at which the outputs are labelled. For outputs labelled as low or labelled as middle by being in a leaf that is not pruned away, the level at which they are labelled is at most $d \leqq \log_2 m$. (For outputs labelled as low, it is in most cases substantially less than this, but we shall not attempt to exploit this effect, since later optimizations will render it negligible.) For outputs labelled as middle because their impurity exceeds $\frac{1}{2}$, the level at which they are labelled is two more than the number of steps taken by the corresponding walk to reach position 0 for the first time. (Again, the first two levels of the tree do not correspond to steps of the random walk.)

In the present instance, this average number of steps can be calculated explicitly and is $4/\sqrt{5} = 1.788 \cdots$. It is obtained from the power series $f(x)^2$ that counts the walks by evaluating $x\,d(f(x)^2)/dx$ at $x = \frac{1}{2}$ or, equivalently, evaluating $f'(x)f(x)$ at $x = \frac{1}{2}$. This evaluation is most conveniently accomplished by dividing the equation $xf(x)^3 - f(x) + x = 0$ by x, differentiating with respect to x, multiplying by x^2, solving for $f'(x)$ in terms of $f(x)$ and x, multiplying by $f(x)$, and evaluating the result at $x = \frac{1}{2}$. Taking account of the equation $f(\frac{1}{2})^2 = (3 - \sqrt{5})/2$, derived earlier, yields the stated result.

We can now sum the contributions to the size of the strong partial classifier. The l outputs labelled as low contribute at most $(s_0 l/2) \log m$, and those labelled as high contribute equally. The outputs labelled as middle by being in a leaf that is not pruned

away contribute at most $Fm^{1-e}\log_2 m$ for some constants F and $e>0$, and the outputs labelled as middle because their impurity exceeds $\frac{1}{2}$ contribute at most Gm for some constant G. Since the $l=\Omega(m)$, we conclude that for every $\varepsilon>0$ and all sufficiently large m (depending on ε), there exists a strong partial classifier with $2m$ inputs, l outputs labelled as low and an equal number labelled as high, and size at most $(s_0+\varepsilon)l\log_2 l$. It follows from Lemma 4.1 that for every $\varepsilon>0$ and all sufficiently large n (depending on ε), there exists a classifier with n inputs and size at most $(s_0/2+\varepsilon)n\log_2 n$. The remainder of this paper is devoted to refining the construction just given to reduce the constant $s_0/2$ to 2.

6. Refined classification trees. If we ask what properties were essential to the construction in the preceding section, we find three. First, the probability that the random walk ever reaches the position 0 is strictly less than 1. Second, the probability that the walk is near position 0 after t steps decreases exponentially with t. Third, the expected number of steps needed to reach 0 (with no contribution from walks that never reach 0) is finite.

The second property is a consequence of the random walk being positively biased. Thus it is natural to seek ways to reduce the number of comparators while preserving the property that the corresponding random walk is positively biased. When this is done, the explicit calculations by which we established the first and third properties will no longer be feasible, but we will see that these properties are consequences of the second property.

The property that the random walk was positively biased follows from the inequality $\vartheta<\frac{1}{4}$ (so that the geometric mean of the factors 2 and 2ϑ, by which impurities change from parent to child, is less than 1). We shall arrange for ϑ to vary from level to level in such a way that the average (again in the sense of the geometric mean) of ϑ is strictly less than $\frac{1}{4}$, but by a very small margin. We shall also exploit the fact that for most of the approximate classifiers in the classification tree, the number of bad elements is very small, so that we may substitute weak approximate classifiers (as defined in § 3).

Let h be a positive integer. Set $\vartheta_h=2^{1/h}/4$ and $\beta_h=(1-\vartheta_h)/\vartheta_h$. Since $\vartheta_h>\frac{1}{4}$, we have $\beta_h<3$. Corollary 2.3 and Lemma 3.1 establish the existence of $\alpha_h>0$ such that, for all sufficiently large n (depending on h), there exists an α_h-weak ϑ_h-classifier with $2n$ inputs and depth 4.

Let us now define a *gadget* to be a tree comprising h levels of α_h-weak approximate classifiers, which therefore approximately classifies the values assigned to its inputs into 2^h classes. The α_h-weak approximate classifier at the root will have tolerance $\frac{1}{8}$, and those at the remaining $h-1$ levels will have tolerance ϑ_h.

Suppose that at most a fraction η of the values assigned to the inputs of the gadget are bad, where $\eta\leqq\alpha_h$. We may determine the impurities of the 2^h sets of outputs by proceeding through successive levels of the tree as before. The root multiplies the impurity by a factor of 2 or $\frac{1}{4}$, and every other node multiplies it by a factor of 2 or $2^{1/h}/2$. A simple calculation shows that the geometric mean of the impurities of the 2^h sets of outputs is $(\frac{1}{2})^{1/2h}$, which is less than 1.

Since all changes to impurities are by factors of $2^{1/h}$, we may consider a random walk on the integers by taking the position to be h times the negative of the logarithm (to base 2) of the impurity, and letting the probability distribution for each step correspond to the changes to impurities for a gadget. The expectation for a step is h times the negative of the logarithm of the geometric mean of the changes, which is $\frac{1}{2}$. Thus the random walk is positively biased.

Since every path through a gadget passes through one weak approximate classifier with tolerance $\frac{1}{8}$ and through $h-1$ with tolerance $\vartheta_h > \frac{1}{4}$, we can construct a gadget with depth at most $4(h-1)+s_0$.

The gadget we have constructed is composed from α_k-weak approximate classifiers, and thus is only useful when the number of bad values is small. We shall call the gadgets described above *cheap* gadgets. We shall also define *dear* gadgets, which have the same tree structure, but with approximate classifiers (rather than weak approximate classifiers) at all nodes, and with all classifiers having tolerance $\frac{1}{8}$. We shall use the same step probability distribution for dear gadgets that we used for cheap gadgets (though of course dear gadgets do much more). The depth of a dear gadget is at most hs_0.

We shall now construct a refined classification tree using gadgets rather than approximate classifiers as nodes below the root and its children (so that the tree is 2^h-ary rather than binary below the first two levels). We assign impurities to each set of outputs of each gadget as before. We shall use a cheap gadget when the impurity of the set of inputs is at most α_h, and a dear gadget when the impurity exceeds α_h.

We shall label outputs as low, high, or middle, and prune away subtrees as before. It remains to estimate the number of outputs labelled as low, high, or middle, and to estimate the size of the resulting strong partial classifier.

The fraction of the outputs that are labelled as middle because their impurity exceeds $\frac{1}{2}$ can be bounded as before by the probability that the corresponding random walk reaches a nonpositive position. (We must consider nonpositive positions, rather than just the position 0, because a single step may now decrease the position by more than 1.) For this purpose we shall use the following lemma.

LEMMA 6.1. *Consider a positively biased random walk starting at a positive position. Then the probability that the walk ever reaches a nonpositive position is strictly less than* 1.

Proof. By Lemma 5.1, the probability that the walk is at a nonpositive position at time t is at most Ab^t for some constants A and $b<1$. The series $\Sigma_{t\geqq 1} Ab^t$ converges, so we may choose T sufficiently large that $\Sigma_{t\geqq T} Ab^t \leqq \frac{1}{2}$. Let $-\Delta$ denote the most negative step that is taken with positive probability, and let $P>0$ denote the probability that a step is positive. Then $Z_{\Delta T} \geqq \Delta T$ with probability at least $P^{\Delta T}$. If this event occurs, the walk cannot reach a nonpositive position in fewer than T additional steps, and the probability of it reaching a nonpositive position in T or more additional steps is no larger than the probability of reaching a position at most ΔT in T or more additional steps, and this is at most $\frac{1}{2}$. Thus the probability of ever reaching a nonpositive position is at most $(1-P^{\Delta T})+P^{\Delta T}/2<1$. □

Using Lemma 5.1 as before, we can again show that the fraction of the outputs that are labelled as middle by being in a leaf that is not pruned away is at most Cm^{-e}, where C and $e>0$ are constants.

To estimate the size of the strong partial classifier we have constructed, we again begin by considering the average level at which outputs are labelled as middle because their impurity exceeds $\frac{1}{2}$. This is bounded by the expectation for the corresponding random walk of the number of steps needed to reach a nonpositive position (with no contribution from walks that never reach a nonpositive position). We shall use the following lemma.

LEMMA 6.2. *Consider a positively biased random walk starting from a positive position. Let U denote the number of steps needed to reach a nonnegative position, if the walk ever reaches a nonpositive position, or* 0 *if the walk never reaches a nonpositive position. Then the expectation of U is finite.*

Proof. Applying Lemma 5.1 with c the negative of the starting position, we have that the probability of being at a nonpositive position at time t is at most Ab^t for some constants A and $b<1$. The convergent series $\Sigma_{t\geqq 1} Ab^t t$ bounds the sum over all visits to nonpositive positions of the times at which the visits occur. This in turn bounds the sum over first visits, which is exactly U. □

For refined classification trees, we shall need an additional estimate concerning the total size of dear gadgets. Since a dear gadget is used precisely when the impurity exceeds a threshold α_h, then the total size of dear gadgets can be bounded in terms of the average time spent by the random walk at positions less than a corresponding constant $c_h = -h \log_2 \alpha_h$. By Lemma 5.1, this average time is bounded by a convergent series $\sum_{t\geqq 1} Ab^t$, for some constants A and $b<1$, and thus is finite. It follows that the total size of dear gadgets is at most Hm, for some constant H.

Summing the contributions to the size as before, we find that for every h and all sufficiently large m (depending on h), there exists a strong partial classifier with $2m$ inputs, l outputs labelled as low and an equal number labelled as high, and size at most $(4(h-1)+s_0+1)l \log_{2^h} l = (4+(s_0-3)/h)l \log_2 l$. Letting h tend to infinity, we see that for every $\varepsilon>0$ and all sufficiently large m (depending on ε), there exists a strong partial classifier as above with size at most $(4+\varepsilon)l \log_2 l$. It follows from Lemma 4.1 that for every $\varepsilon>0$ and all sufficiently large n (depending on ε), there exists a classifier with n inputs and size at most $(2+\varepsilon)n \log_2 n$. Thus we have achieved the goal of this paper.

7. Embellishments. In this section we shall offer some additional comments on the embellishments to our main result that were mentioned in the introduction.

If one wishes to classify n values into classes of cardinality t and $n-t$, this can be accomplished by modifying the definitions of some of the components in the construction. One redefines "strong partial classifier" so that the numbers of outputs labelled as low and high are in the correct proportion, t: $n-t$, and so that $t/2$ and $(n-t)/2$ values are eligible to appear as low and high outputs, respectively. One must then modify the upper levels of the classification trees to accommodate the new definitions of "good" and "bad" values, which now differ on the two sides of the trees. (Special care is necessary if t or $n-t$ is much smaller than n.) The constant factor in the final result is independent of t, but this should be regarded as a deficiency rather than a merit, since it is to be expected that this constant should decrease as t is varied away from $n/2$.

The problem of obtaining an explicit construction is attacked by replacing Lemma 2.1 by an analogous explicit result (see Pippenger [Pi, § 3.2]). A complication arises from the fact that the explicit results hold only for certain n, rather than for all sufficiently large n. Thus one obtains, for example, weak approximate classifiers with $2(q+1)$ inputs, where q is a prime congruent to 1 modulo 4. In constructing classification trees, one must always settle for using the next smaller weak approximate classifier of this form, and reconcile oneself to labelling the remaining outputs of the parent approximate classifier as middle. Standard results on the distribution of primes, however, allow one to show that only a negligible fraction of the outputs are labelled as middle in this way. Using weak approximate classifiers with depths 8, 12, and 30 results in a final bound slightly less than $6n \log_2 n$.

Finally, if one wishes to ensure that the depth of the final classifier is $O(\log n)$, one must modify the recursive construction so that the various strong partial classifiers are "overlapped" in depth. To do this, it is most convenient to make them "stronger" (so that $m/4$ rather than $m/2$ values are eligible to appear at an output labelled as

low or high). One can then show that enough outputs are available from the shallower levels of the first r strong partial classifiers to supply inputs for the $(r+1)$st strong partial classifier.

8. Conclusion. We have established the existence of classifiers with many fewer comparators than those previously known. For most constructions that rely on expanding graphs (such as those given by Bassalygo [B]), the constant factor depends on the current state of technology of expanding graphs, and can be expected to improve with further advances in the state of this art. The result of this paper, however, will not be improved in this way: the constant in the leading term of the size depends only on the degree needed for expanding graphs to expand very small sets, and this aspect of expanding graphs is understood completely (graphs of degree s can expand small sets by any factor up to $s-1$, but not by more).

It would be of interest to see if a similar situation can be brought about for other applications of expanding graphs, perhaps even for the most celebrated application of all, the sorting networks of Ajtai, Komlós, and Szemerédi.

REFERENCES

[Aj] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *Sorting in c* log *n parallel steps*, Combinatorica, 3 (1983), pp. 1-19.

[Al] V. E. ALEKSEEV, Sorting algorithms with minimum memory, Kibernetica, 5 (1969), pp. 99-103.

[B] L. A. BASSALYGO, *Asymptotically Optimal Switching Circuits*, Problems Inform. Transmission, 17 (1981), pp. 206-211.

[Be] S. W. BENT AND J. W. JOHN, *Finding the median requires* $2n$ *comparisons*, in Proc. 17th ACM Symposium on Theory of Computing, 1985, pp. 213-216.

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

[Pa] M. S. PATERSON, *Improved sorting networks with* $O(\log n)$ *depth*, Algorithmica, to appear.

[Pi] N. PIPPENGER, *Communication Networks*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., North-Holland, Amsterdam, 1990.

[S] A. SCHÖNHAGE, M. PATERSON, AND N. PIPPENGER, *Finding the Median*, J. Comput. System Sci., 13 (1976), pp. 184-199.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 888-910, October 1991
© 1991 Society for Industrial and Applied Mathematics
006

AN OUTPUT-SENSITIVE ALGORITHM FOR COMPUTING VISIBILITY GRAPHS*

SUBIR KUMAR GHOSH† AND DAVID M. MOUNT‡

Abstract. The visibility graph of a set of nonintersecting polygonal obstacles in the plane is an undirected graph whose vertex set consists of the vertices of the obstacles and whose edges are pairs of vertices (u, v) such that the open line segment between u and v does not intersect any of the obstacles. The visibility graph is an important combinatorial structure in computational geometry and is used in applications such as solving visibility problems and computing shortest paths. This paper presents an algorithm that computes the visibility graph of a set of obstacles in time $O(E+n \log n)$, where E is the number of edges in the visibility graph and n is the total number of vertices in all the obstacles.

Key words. visibility graph, output-sensitive algorithms, shortest paths

AMS(MOS) subject classifications. 68Q25, 68U05

1. Introduction. The *visibility graph* of a set of nonintersecting polygonal obstacles in the plane is a graph whose vertex set consists of the vertices of the obstacles and whose edges are the pairs of vertices (u, v) such that the open line segment between u and v does not intersect any of the obstacles. In this paper an output-sensitive algorithm is presented for computing the visibility graph of a set of polygonal obstacles. The visibility graph is a fundamental combinatorial structure in computational geometry; it is used, for example, in applications such as computing shortest paths amidst polygonal obstacles in the plane [11]. In particular, given a set of polygonal obstacles in the plane, the shortest-length path between any two points s and t travels along the edges of the visibility graph of the obstacle set augmented with the points s and t [10], [15].

In the worst case the visibility graph of a set of obstacles with n total vertices may contain $O(n^2)$ edges. An $O(n^2 \log n)$ algorithm for this problem was given by Lee [9] and Sharir and Schorr [15]. Later, worst case optimal $O(n^2)$ algorithms were discovered by Asano et al. [1] and Welzl [16]. If the visibility graph contains relatively few edges, for example, when there are many densely packed objects, it is desirable to have an algorithm whose running time is a function of the number of edges. Hershberger has described an output-sensitive algorithm for the case of computing the visibility graph within a simple polygon (once the polygon has been triangulated) [6], and Overmars and Welzl have given an algorithm for computing the visibility graph for a set of disjoint polygonal obstacles whose running time is $O(E \log n)$ and whose space is $O(n)$, where E is the number of edges in the visibility graph [14].

In this paper we present an algorithm that computes the visibility graph of an arbitrary set of disjoint obstacles with running time $O(E+n \log n)$. The $O(n \log n)$ term is overhead needed for computing a particular triangulation of the obstacle-free

* Received by the editors July 19, 1989; accepted for publication (in revised form) December 11, 1990. A preliminary version of this paper appeared in the Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 11-19.

† Computer Science Group, Tata Institute of Fundamental Research, Bombay, India. The work of this author was supported by Air Force Office of Scientific Research grant AFOSR-86-0092, while he was visiting the Center for Automation Research at the University of Maryland, College Park, Maryland.

‡ Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742. The work of this author was supported by National Science Foundation grant CCR-8908901.

space, after which the algorithm runs in $O(E)$ time. This is optimal in the worst case with respect to E and n, since there are cases of n obstacles where $E = O(n)$, but computing the visibility graph is equivalent to sorting a set of n points [1]. The algorithm uses $O(E+n)$ space. Recalling the application of computing shortest paths amidst polygonal obstacles, once the visibility graph has been constructed, the shortest path can then be computed in $O(E+n \log n)$ time by Fredman and Tarjan's variation of Dijkstra's algorithm using Fibonacci heaps [2].

The key to the algorithm's efficiency is a way of structuring the edges of the visibility graph in terms of a set of objects called *funnel sequences.* Intuitively, the funnel sequence associated with an edge of an obstacle encodes the set of vertices that can see some portion of this edge. We present a novel technique of traversing the funnel sequences.

Throughout, P will denote a bounded polygonal domain, which we will think of as a simple polygon (forming the *external boundary*) whose interior contains a set of simple polygons (the *holes*) such that the holes have pairwise disjoint interiors. Define the *free space* to be the closed space lying on or within the external boundary and on or outside the holes. (If no exterior boundary is given, the convex hull of the obstacle set, which is computable in $O(n \log n)$ time, suffices as the external boundary.) Throughout, in using the term *polygonal domain*, we will assume the existence of an external boundary. The *boundary* of a polygonal domain is represented by a single counterclockwise cycle of directed edges forming the holes. Thus for each directed edge, free space lies to the left side of the edge. To simplify the presentation, we will make the "general position" assumptions throughout that no three vertices of the polygonal domain are collinear and no two vertices share the same x-coordinates. We will also assume that each vertex of P is incident on exactly two edges of P (line segment obstacles can be handled as degenerate polygons with two oppositely directed edges). Our results hold in the absence of these assumptions, but the presentation would be complicated by a number of tedious special cases that would need to be considered.

2. The plane-sweep triangulation. As mentioned in the Introduction, the visibility graph algorithm is based on a triangulation of free space. Let $T_1, T_2, \cdots, T_m$ denote the triangles of this triangulation. Thus the free space region defined by P is just the union of these triangles: $P = \bigcup_{i=1}^{m} T_i$. The visibility graph algorithm operates by constructing a series of subsets of free space by successively adjoining triangles to one another, $P_1 = T_1$, $P_2 = T_1 \cup T_2$, $P_3 = T_1 \cup T_2 \cup T_3$, etc. We compute a complete visibility graph for each subset P_k by augmenting the visibility graph for P_{k-1}. (To be exact, we simultaneously add a number of triangles incident on a single vertex.) Because of the nature of the augmentation procedure, it will be important to select the triangulation and the ordering of triangles in a careful way. Fortunately, there is a simple and natural triangulation based on plane-sweep which suffices for our purposes. The triangulation algorithm is essentially equivalent to one described by Mehlhorn [13, pp. 160–172]. Although Mehlhorn's algorithm assumes that the polygon has no holes, the algorithm generalizes easily.

The idea behind the plane-sweep triangulation for polygons is most easily illustrated by describing the plane-sweep triangulation of a set of points $p_1, p_2, \cdots, p_n$. As is common in plane-sweep algorithms, first the points are sorted in increasing order of their x-coordinates. The triangulation initially contains no edges, just the vertex whose x-coordinate is minimum. Inductively, let us assume that the first $k-1$ points have been triangulated. Think of the outer boundary of the triangulated region as a

polygon P_{k-1}, namely the convex hull of the first $k-1$ points. Clearly the point p_k lies outside of P_{k-1}. Thus we can incorporate p_k into the triangulation by connecting p_k to all of the points on the boundary of P_{k-1} that are visible from p_k (thinking of P_{k-1} as an obstacle). The point p_k will be joined to an inward-convex chain of vertices on the boundary of P_{k-1}.

The plane-sweep triangulation of the interior of a polygonal domain is similar. First the vertices are sorted by x-coordinate. Let $v_1, v_2, \cdots, v_n$ denote the resulting sequence. Inductively assume that the first $k-1$ vertices have been incorporated into the triangulation. The outer boundary of the triangulated region consists of a set of disjoint simple polygons, which may degenerate to isolated points and line segments. Thinking of the edges of the polygonal domain as forming obstacles, the vertex v_k is incorporated into the triangulation by adding visible segments between v_k and all its visible neighbors on the boundary of the triangulated region.

The plane-sweep triangulation can be built in $O(n \log n)$ time. The important property of the plane-sweep triangulation, which will be exploited by our algorithm, is summarized in the next lemma. This lemma follows from the discussion in [13]. See Fig. 1.

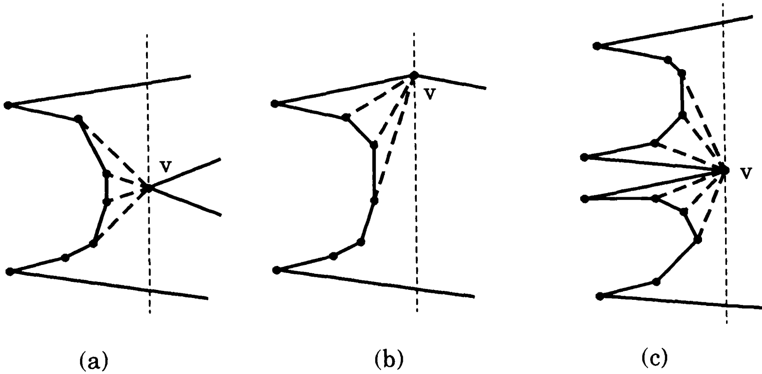


FIG. 1. *Connecting a point to the triangulation.*

LEMMA 2.1 (Mehlhorn [13]). *Consider the triangles formed as an arbitrary vertex v is incorporated into the triangulation of a polygonal domain P. These triangles form either one or two connected sequences about v such that the sides opposite v form an inward-convex chain with respect to v (degenerating possibly to a single point). If there are two such sequences, then these sequences are separated from one another by the boundary of P (Fig. 1(c)).*

3. The funnel sequence. The visibility graph of a polygonal domain possesses a great deal of structure when seen within the context of the polygon itself. In this section we describe the fundamental structure that our algorithm manipulates, called the *funnel sequence* for an edge of the polygonal domain P. Funnels arise naturally in shortest-path and visibility problems in simple polygons [5], [6], [10]. We begin with some definitions and observations about funnels.

Define a *visible chain* in polygonal domain P to be a path in the visibility graph of P. To avoid confusion, we will use the term *edge* when referring to an edge of a polygon, and the term *visible segment* or just *segment* when referring to an edge in a visibility graph. A chain is *convex* if the figure defined by joining the two endpoints

of the chain is a convex body. Consider a vertex v that is visible from an interior point z of an edge (x, y) of P. For the sake of illustration, imagine that the edge (x, y) is directed upwards and point v is to the left of the edge (see Fig. 2). Define the *lower chain* of v with respect to (x, y) to be the unique convex visible chain from v to x such that the interior region bounded by this chain and by the line segments vz and xz is empty. Intuitively, the lower chain is formed by imagining that the segment vz is a rubber band and sliding the point z of this rubber band down the edge (x, y) until reaching x. The upper chain of v with respect to (x, y) is defined analogously for y. The lower chain, upper chain, and edge (x, y) bound a simple polygon in P which we call a *funnel*.

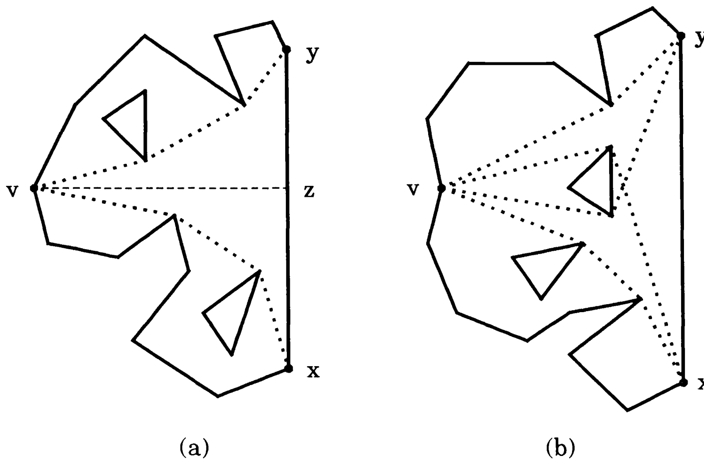


FIG. 2. *Visible chains and funnels.*

By definition, the interior of the funnel contains no vertices and no edges of P. The vertex v is called the *apex* of the funnel and the edge (x, y) is the *base* of the funnel. Unlike funnels that arise in simple polygons [6], in polygonal domains there may be many funnels sharing a single apex vertex (see Fig. 2(b)). We will think of these apexes as being distinct objects occupying the same physical location in space.

Considering the visibility graph for P and a vertex v of P. Let $u_0, u_2, \cdots, u_m$ be the clockwise sequence of vertices that are visible from v so that (u_0, v) and (v, u_m) are edges of P. For every pair of cyclically adjacent vertices u_{i-1} and u_i, there is a unique edge e of the polygonal domain that can be seen by an observer located at v looking between these vertices (for otherwise, there would be another visible vertex between them). Thus there is a unique funnel whose apex is v, whose base is e, whose upper chain begins with (v, u_{i-1}), and whose lower chain begins with (v, u_i). Given the first directed segment (v, u_i) of the lower chain, the first directed segment (v, u_{i-1}) of the upper chain is uniquely determined, and vice versa. An immediate result of this correspondence is the following.

LEMMA 3.1. *There is a* 1-1 *correspondence between pairs of cyclically adjacent directed segments of the visibility graph about a vertex* v, $((v, u_{i-1}), (v, u_i))$ *for* $0 < i \leqq m$, *and the funnels whose apex is* v.

COROLLARY. *The total number of funnels in a visibility graph with* E *undirected edges and* n *vertices is* $2(E-n)$, *which is* $O(E)$.

For a given edge (x, y) of the polygonal domain P, let FNL(x, y) denote the set of funnels whose base edge is (x, y). Recall that the interior of P lies to the left of the edge, so these funnels all lie on the left of (x, y). For completeness, vertices x and y can each be thought of as the apexes of degenerate funnels in FNL(x, y). (There are $2n$ degenerate funnels, so this does not alter the number of funnels asymptotically.) If v is the apex of a funnel in FNL(x, y), and u is the first vertex on the lower chain from v to x, then u is visible from the edge (x, y) implying (by convexity of funnels) that u is the apex of a unique funnel that is contained within v's funnel. If we think of the apex u as the parent of the apex v, we see that the set of funnels in FNL(x, y) forms a tree rooted at x whose paths are the lower chains of FNL(x, y). Note that it is important to distinguish vertices from apexes here because the same vertex can appear many times as an apex in FNL(x, y), whereas each apex can appear only once. Each path from a node to the root of this tree is a convex visible chain that turns clockwise. Call this the *lower tree* for the edge (x, y) (see Fig. 3(a)). Analogously, we define the *upper tree* to consist of the tree of upper chains of FNL(x, y) rooted at y (see Fig. 3(b)). Paths from a node to the root in the upper tree are convex visible chains that turn counterclockwise.

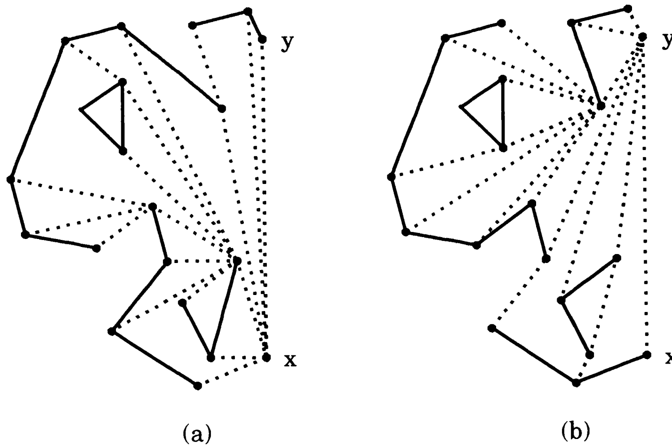


FIG. 3. *The lower and upper trees.*

We can define a natural linear ordering on the funnels of FNL(x, y) based on these trees by considering the clockwise preorder traversal of the lower tree. There is another natural order that results from considering a clockwise postorder traversal of the upper tree. In both orderings, the degenerate funnel at x is first, and the degenerate funnel at y is last. Our next result states that these orders are in fact equal to one another. We refer to this clockwise ordering of funnels as the *funnel sequence* for the edge (x, y).

LEMMA 3.2. *The linear orders on* FNL(x, y) *arising from a clockwise preorder traversal of the lower tree and a clockwise postorder traversal of the upper tree are the same.*

Proof. Let f_1 and f_2 be two funnels so that f_1 precedes f_2 in a clockwise preorder traversal of the lower tree. Think of the lower chains of f_1 and f_2 as paths from the root x to the apexes of these chains, and think of upper chains as paths from y. There are two reasons that f_1 may precede f_2: (1) the lower chain of f_1 is a subchain of the lower chain of f_2 and (2) the lower chain of f_2 diverges clockwise from f_1's lower chain at some common ancestor.

In case (1) the apex of f_1 lies on the lower chain of f_2. By the emptiness of funnel f_2, the upper chain of f_1 contains a single segment that lies entirely within f_2 and joins the apex of f_1 to a vertex v on the upper chain of f_2 (either at a point of tangency or at y). See Fig. 4(a). This implies that the upper chain for f_2 diverges clockwise from the upper chain for f_1 at v, and hence f_1 precedes f_2 in any clockwise traversal of the upper tree.

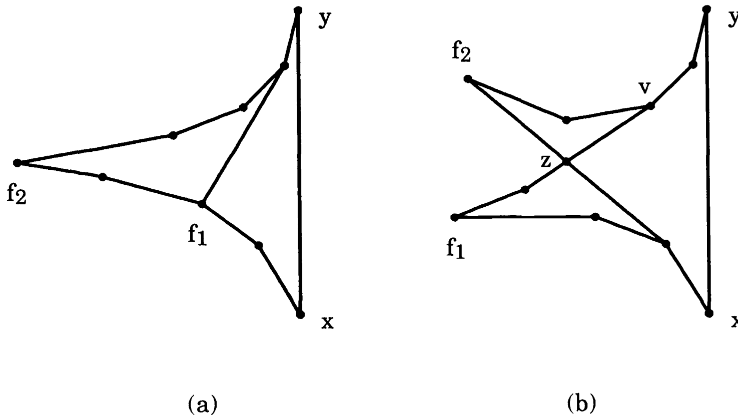


FIG. 4. *Funnel ordering.*

In case (2) f_2's path diverges from f_1 along a segment that enters into the interior of f_1. Since funnels are empty, this segment must eventually intersect the boundary of f_1 at some point z (which may or may not be a vertex). The point z must lie on the upper chain of f_1, since it cannot intersect the interior of edge (x, y), and by convexity it cannot cross the lower chain of f_1. See Fig. 4(b). If z is the apex of f_2, implying that f_2 is an ancestor of f_1 in the upper tree, then f_1 precedes f_2 in any postorder traversal of the upper tree. Otherwise the lower chain of f_2 crosses the upper chain of f_1 at z. The upper chain of f_2 cannot enter into the interior of f_1 by the emptiness of funnels, and hence the upper chain of f_2 must diverge clockwise from the upper chain f_1 at some vertex v before reaching z. This implies that the upper chain for f_1 precedes f_2 in any clockwise traversal of the upper tree. □

4. The enhanced visibility graph. In this section we describe the basics of the visibility graph algorithm. We assume that we have computed the plane-sweep triangulation for the polygonal domain P. (Actually, the process described here could be performed while the triangulation is being built.) Recall from § 2 that the vertices $v_1, v_2, \cdots, v_n$ of P have been sorted in increasing order by x-coordinate, and they are incorporated into the triangulation in this order. Let P_k denote the triangulated region containing the vertices $v_1, \cdots, v_k$. We will think of P_k as a polygonal domain contained within P (it may be disconnected and contain isolated points and edges).

For each k we maintain a structure called the *enhanced visibility graph* for P_k. Before specifying the enhanced visibility graph we first give some definitions. Consider a vertex v in the visibility graph and consider the visible segments directed out of v. This list will include the two boundary edges of P incident on v. Let (v, u) be a visible segment incident on v. Define the *clockwise successor* of (v, u), CW (v, u), to be the next visible segment about v in clockwise order and define the *counterclockwise successor* of (v, u), CCW (v, u), analogously. Define the *clockwise extension* CX (u, v) of a visible

segment directed into v as follows (note the reversal of arguments). Rotate the ray from v through u clockwise by 180 degrees about v. If this sweep lies entirely within the interior of P locally about v, then the extension is the very next visible segment encountered after the 180 degree sweep (by our assumption of the noncollinearity of three vertices, there will be no segment at exactly 180 degrees). If not, then the clockwise extension is undefined. The *counterclockwise extension*, CCX (u, v), is defined symmetrically using a counterclockwise sweep. Fig. 5 illustrates three of these entities, and the fourth, CX (u, v), is undefined for this example. Finally, define REV (u, v) to be the directed reversal (v, u).

DEFINITION. Define the *enhanced visibility graph* for polygon P to consist of:

• the boundary of P represented such that the two neighbors of a given vertex can be found in constant time;

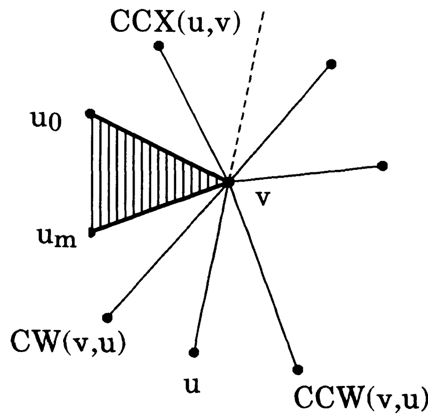


FIG. 5. *Traversal primitives.*

• the visibility graph for P, represented such that the operations CCW, CW, CCX, CX, and REV can be evaluated in constant time each; and

• the funnel sequence FNL (x, y), for each edge (x, y) on the boundary of P, represented (say, as a doubly linked list) so that the operations of split, concatenate, predecessor, and successor can be performed in constant time each. (To be exact, our algorithm only maintains the funnel sequence for a selected set of boundary edges along the right side of P, along which we will augment the triangulation.)

In § 7 we will show how to implement CW, CCW, CX, CCX, and REV, but for now we assume that these operations are available to us. From Lemma 3.1 we may assume that each funnel apex is uniquely represented by giving the first segment in its lower chain (directed out of the funnel's apex), but we will refer to apexes by vertices, when the funnel is clear from context. Next we observe that the enhanced representation of the visibility graph contains sufficient information to permit traversals of the upper and lower trees.

LEMMA 4.1. *Consider the enhanced visibility graph of a polygonal domain P, and suppose that (u, v) is any directed segment of the lower tree of an edge (x, y) of P, such that u is a parent of v. The following relatives of u and v in the lower tree can be computed in constant time*:

(i) *the parent of u,*

(ii) *the extreme clockwise and counterclockwise children of v, and*

(iii) *the clockwise and counterclockwise siblings of v.*

Analogous claims hold for the upper tree.

Proof. We prove the lemma for lower trees, and a symmetric argument establishes the result for upper trees. For (i), by the clockwise turning of the lower chains, the parent of u in the lower tree is the apex whose lower chain begins with the head vertex of the clockwise extension CX (v, u), that is, CX (REV (u, v)), provided it exists (see Fig. 6). If this extension is undefined, then it follows (by the emptiness of funnels) that $u = x$, and hence u is the root of the tree.

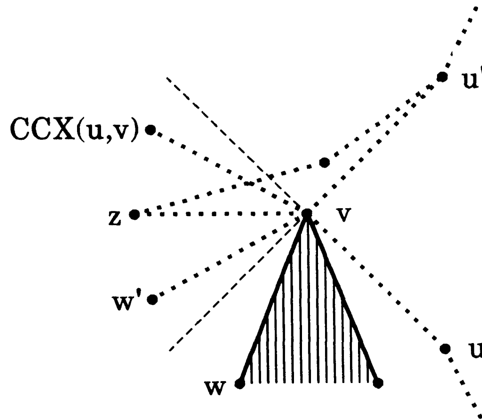


FIG. 6. *Tree traversal.*

Since (u, v) is a segment of the lower tree for (x, y), v is the apex of a funnel that is visible from the interior of the edge (x, y). The first lower chain segment of this funnel is (v, u) and the first upper chain segment is CCW (v, u). Let $(v, u') =$ CCW (v, u). To establish (ii), let (w, v) be the directed edge on the polygon's boundary, such that the interior of the polygon lies to the left of this directed edge. If the counterclockwise extension CCX (u, v) is undefined (implying that w lies in the halfplane to the right of segment (u, v)), then, by the convexity of lower chains, v cannot have a child in the lower tree, and hence is a leaf. Otherwise, any children of v must lie between CCX (u, v) and (v, w) counterclockwise about v (see Fig. 6). Let z be such a vertex. For z to be a child of v, the funnel with apex z whose lower chain begins with segment (z, v) must be visible from the interior of edge (x, y). This is true if and only if the counterclockwise angle $u'vz$ is less than 180 degrees. (The "only if" part of this statement is true from the convexity of the upper chains. The "if" part holds because if $u'vz$ is less than 180 degrees, then the ray from z through v passes through the interior of the funnel whose apex is v and strikes the interior of the edge (x, y).)

Let w' be chosen such that if the counterclockwise extension CCX (u', v) exists, then $(v, w') =$ CW(CCX (u', v)), and otherwise $w' = w$. Clearly, w' is computable in constant time, and it follows from the previous discussion that the children of v are exactly those apexes z visible from v that lie counterclockwise from the head of CCX (u, v) to w', assuming that this angular sector is not empty. If so, the reversal of these two edges, REV(CCX (u, v)) and REV (v, w'), are the first edges of the lower chains of the extreme clockwise and counterclockwise children of v, respectively. If the sector is empty, then v is a leaf.

For (iii), note that the clockwise sibling of v is just CW (u, v), provided that v is not the extreme clockwise child of u. A symmetric statement holds for the counterclockwise sibling of v. By (ii) we can test whether v is an extreme child of u. □

COROLLARY. *Given the enhanced visibility graph, clockwise and counterclockwise traversals of the lower and upper trees can be performed in time proportional to the sizes of the trees, and a funnel can be traversed in time proportional to its size.*

5. Splitting the funnel sequence. Our next task is to describe how to use the ability to traverse the enhanced visibility graph in order to add a new vertex into the visibility graph.

The basic loop of the visibility graph algorithm consists of successively adding triangles from the triangulation of the polygonal domain and updating the visibility graph with each addition. We assume inductively that an enhanced visibility graph has been computed for the interior of the triangulated region so far. For each new triangle added, we update the visibility graph appropriately. The fundamental operation on which our algorithm is based is procedure SPLIT. This procedure is given an enhanced visibility graph for a polygonal domain P, a directed edge (x, y) on the external boundary of P, and a point v lying to the right of this edge so that the triangle xvy is external to P. The procedure essentially merges the triangle xvy into P (erasing the edge (x, y)) and computed the enhanced visibility graph of the resulting polygonal domain.

After the edge (x, y) is removed, every vertex in P that was visible from some interior point of the edge (x, y) will be visible from either the interior of edge (x, v) or edge (v, y) or both. The apex of a funnel of P is visible from both edges (x, v) and (v, y) (through the funnel) if and only if the apex is visible from v. Consider a funnel with apex u, whose upper chain is U and whose lower chain is L. (We will often refer to a funnel by giving the name of the vertex that is its apex whenever the actual funnel is clear from context.) If u can see v through the funnel, then SPLIT will add the visible segment between u and v, in effect splitting the funnel u into two funnels, one for FNL (x, v) whose lower chain consists of L and whose upper chain has only the segment (u, v), and one for FNL (v, y) whose upper chain consists of U and whose lower chain has only the segment (u, v) (see Fig. 7(a)). If u can see only one edge through the funnel, say the lower edge (x, v), then SPLIT will make u the apex of a funnel to be added to FNL (x, v). The lower chain of such a funnel will consist of L, and the upper chain will consist of a tangent segment from v to the upper chain U, followed by the remainder of U to u (see Fig. 7(b)).

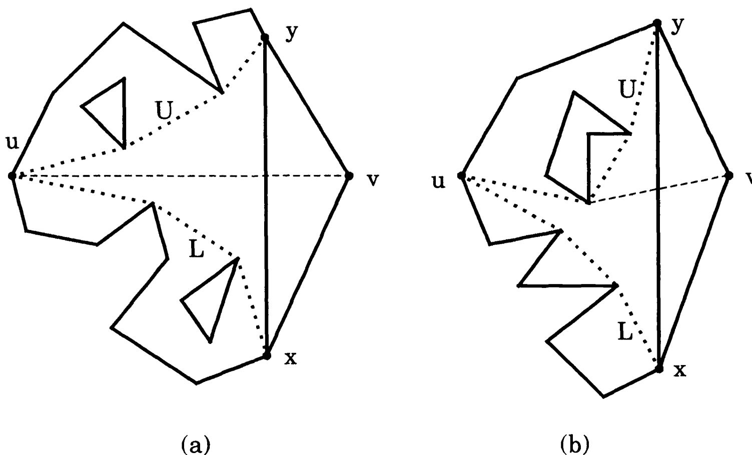


FIG. 7. *Splitting a funnel.*

To illustrate the operation of SPLIT in greater detail, consider two funnel apexes u and t that are visible from v in consecutive clockwise order about v. Between these apexes lies a *pocket* of visibility, where there may exist apexes that can see the edge (x, y), but not the vertex v. Extend the visible segments (v, u) and (v, t) until reaching points q' and r' on the boundary of P (see Fig. 8). There are no vertices or polygonal edges in the triangle $vq'r'$ because there are no visible vertices between u and t. Imagine for the moment two funnels whose apexes are the points q' and r'. These points are visible from the interior of edge (x, y), and hence (as apexes of two funnels that pass between u and t) they can be put into the linear order of FNL (x, y). It is not hard to see that q' and r' will be consecutive in funnel order and (as will be proved in Lemma 5.2) $q' < r'$. (We will use the notation $<$ and $>$ to relate apexes in funnel order.) Let q be the true apex in FNL (x, y) that precedes q' in funnel order, and let r be the true apex in FNL (x, y) that succeeds r'. It may be that $q = u$ or $r = t$. Intuitively, if $q \neq u$, then every apex a, $u < a \leqq q$, has its visibility of v blocked from below by u. (We say an apex q's visibility of v is *blocked from below* by u if the lower chain of q passes through u, and u is a point of tangency with respect to v on this chain.) These apexes are only visible from the upper edge (v, y). Similarly, if $r \neq t$, then every apex a, $r \leqq a < t$, has its visibility of v blocked from above by t. These apexes are only visible from the lower edge (x, v).

The procedure SPLIT operates by finding the funnel apexes that are visible from v in clockwise order. For each consecutive pair of visible apexes that it finds (such as u and t) there is a pocket of edge visible apexes. The procedure locates apexes (such as q and r) at which the pocket can be split. Since the funnel sequence is a simple doubly linked list, the splitting can be done in constant time, once the endpoints of the split are known. The key to the efficiency of the procedure is to locate t, q, and r quickly, once u is known. The heart of the SPLIT procedure is a search of the enhanced visibility graph, which when given a visible vertex u, finds these entities and, in general, a number of other visible vertices in time proportional to the number of visible pairs encountered. Thus the effort of the algorithm will be amortized against the number of newly discovered visible segments.

Before describing the SPLIT procedure, we investigate the deeper structure of the upper and lower trees. The fundamental intuition that we exploit is that within a

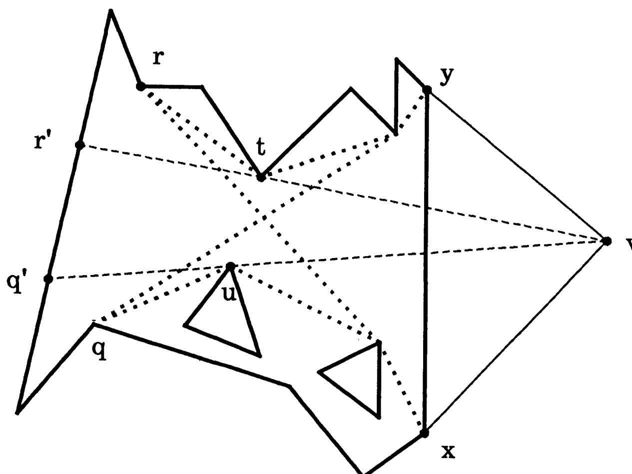


FIG. 8. *Splitting the funnel sequence.*

sufficiently small region, namely two vertices that are adjacent in funnel order, the visibility structure is really no different than the visibility structure of a simple polygon *without* holes. To make this intuition more formal, we begin with a definition. Consider a pair of apexes $q < r$ that are consecutive in the funnel order of the edge (x, y). Define the *hourglass* of q and r to consist of the edge (x, y), the upper chain from r to y, the line segment (r, q), and the lower chain from q to x (see Fig. 9).

LEMMA 5.1. (i) *The four parts of an hourglass do not intersect each other except at their endpoints, and thus they define a closed simple polygon.*

(ii) *The interior of the region bounded by the hourglass is empty, that is, it contains no vertices or edges of P.*

Proof. To prove (i) consider the upper tree for edge (x, y). Since q immediately precedes r in funnel order, by Lemma 3.2, r is the clockwise postorder successor of q in the upper tree. Thus either r is the parent of q in the upper tree or else r is the furthest counterclockwise leaf in the subtree rooted at the clockwise sibling of q. If r is the parent of q in the upper tree, then the hourglass degenerates into the funnel for q, and both parts of the lemma follow immediately. Thus assume that r is not the parent of q, and let s denote the parent of q in the upper tree (see Fig. 9). The upper chain from r to y passes through s. By the clockwise and counterclockwise turning natures of the lower and upper trees, respectively, the line passing through q and s separates the upper chain from r to y from the lower chain from q to x; thus these portions of the hourglass's boundary do not intersect (except at their endpoints). This line also separates the segment (q, r) from the lower chain passing from q to x, implying that these parts of the hourglass boundary do not intersect. A symmetric argument (applied to the lower tree) shows that segment (q, r) does not intersect the upper chain from r to y. Finally, since all these structures lie within P, none of them intersects the edge (x, y).

To show (ii), consider the region R bounded by the portion of the upper chain from r to s, the segment (q, s), and the segment (q, r). Clearly, the region R, together with the interior of the funnel for q, subdivide the interior of the hourglass into disjoint regions. Because q and r are consecutive in the funnel order, there can be no vertices in the interior of R. Furthermore, there can be no edges of P in the interior of R since, in the absence of vertices in the region, such an edge would have to cross either the segment (q, s) or else the upper chain from r to s, but these are formed entirely from

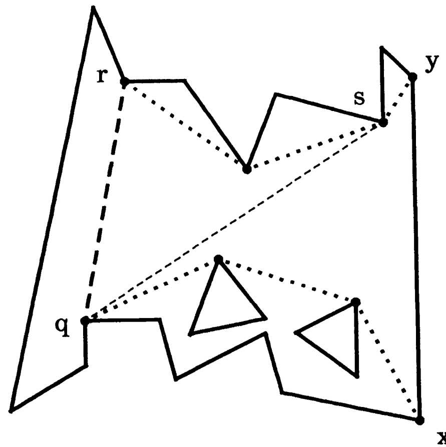


FIG. 9. *The hourglass defined by two consecutive apexes.*

visible segments. This implies that the interior of R is empty. Fact (ii) follows immediately by the emptiness of the funnel for q. □

We will also need the observation that there is consistency between the funnel ordering for edge (x, y) and the new edges.

LEMMA 5.2. *Let (x, y) be an edge of a polygonal domain P, and let v be a point external to P forming an empty triangle with (x, y). Let P' be the polygonal domain which results by replacing the edge (x, y) with the two edges (x, v) and (v, y). Let u and w be two apexes of* FNL (x, y) *such that u precedes w in funnel order.*

(i) *If u and w are both visible from v in P', then u precedes w in clockwise order about v from x to y.*

(ii) *If u and w are not visible from v in P', but both are visible from the lower edge (x, v), then u will precede w in the funnel order of (x, v) (as apexes in* FNL (x, v)*).*

(iii) *If u and w are not visible from v in P', but both are visible from the upper edge (v, y), then u will precede w in the funnel order of (v, y) (as apexes in* FNL (v, y)*).*

Proof. Assertion (ii) holds because in this case the lower chains for all such funnels are unaffected, and thus the tree relationships are preserved. Assertion (iii) holds because in this case, the upper chains for such funnels are unaffected, and thus the tree relationships are preserved (using Lemma 3.2).

To prove (i), we consider the two ways in which u can precede w in the lower tree. If u is an ancestor of w in the lower tree, then the lemma follows immediately from the convexity of the lower chains and the fact that w is visible from v. Otherwise, since u precedes w in funnel order, they share a common ancestor u'' in the lower tree, and the lower chain passing from x to w diverges clockwise from the lower chain from x to u at u''. This implies that the first segment on the path from u'' to w passes into the interior of the funnel for u. Since the funnel for u is empty, this segment must intersect the upper chain for u at some point z (not necessarily a vertex). Since u is visible from v, all of its upper chain is visible from v, and all the points on this upper chain lie clockwise from u about v. Thus z lies clockwise from u. By the convexity of the lower chains, and the fact that w is visible from v, w lies clockwise from z and hence clockwise from u about v. □

We now return to the description of the SPLIT procedure. SPLIT is a recursive procedure that is called under the following conditions. We are given the enhanced visibility graph for a polygonal domain P, an edge (x, y), and a point v external to P forming a triangle with (x, y). Throughout the description, P, x, y, and v will remain constant. We are also given a funnel apex u that is visible from v. Let w be the parent of u in the upper tree. By Lemma 3.2, w follows u in funnel order. By the convexity of the upper chain, it follows that w is also visible from v. We assume that all of the visible segments from (v, x) to (v, u) in clockwise order about v have been added to the visibility graph but that none of the visible segments after this have been added. Let FNL $[u, w]$ denote the subsequence of FNL (x, y) that contains all the funnels (in funnel order) between u and its parent w, noninclusive. (Note that the elements of FNL $[u, w]$ have edge (x, y) as their base, not the segment (u, w).) It is easy to see that, since w is the parent of u in the upper tree, the upper chain of every funnel in FNL $[u, w]$ passes through w (although the lower chain of every funnel in FNL $[u, w]$ need not pass through u).

Recall that funnels are not stored explicitly as upper and lower chains, but rather we only store the first segment of the lower chain and extract all other segments from traversals of the enhanced visibility graph. Thus as segments are added to the enhanced visibility graph, the structure of the funnels changes. With this in mind, on return from the call SPLIT (u, w), the following tasks will be completed.

(1) All the visible segments between v and visible apexes of FNL $[u, w]$ are added (each addition will, in effect, split some funnel into two funnels),

(2) FNL $[u, w]$ is split into two funnel sequences, those with apexes visible only to the lower edge (x, v) and those with apexes visible only to the upper edge (v, y). The first set of funnels are concatenated onto the end of FNL (x, v) and the second set is concatenated onto FNL (v, y). (Note that the addition of the visible segments in (1) implies that all funnels in FNL $[u, w]$ will be in either one class or the other.)

FNL (x, v) and FNL (v, y) are initialized to empty. The algorithm proceeds by first creating the visible segment (v, x), then calling SPLIT (x, y), which does the bulk of the work, and finally adding the visible segment (v, y). The fact that SPLIT will encounter visible pairs in clockwise order about v together with Lemma 5.2 implies that the final order of these newly formed sequences will be correct. We now give an annotated description of the procedure. Throughout the description, unless otherwise noted, all funnels and the lower and upper trees belong to FNL (x, y). Although we will often ignore the distinction between apexes and vertices in the description below, recall that every apex is represented by the first edge of its lower chain.

PROCEDURE SPLIT (u, w):

(1) We begin by searching for the last funnel apex q after u in funnel order whose visibility of v is blocked by u. The path to q in the lower tree may visit many vertices that are invisible from v, so we seek a more efficient route. Our method instead locates the successor r of q in funnel order (see Fig. 10). Since we have added the visible segment (u, v), this segment is the first segment of an apex in the lower tree of FNL (v, y). Let u' be the extreme clockwise child of this apex in the lower tree for (v, y). This is the most clockwise child of u whose visibility of v is blocked by u.

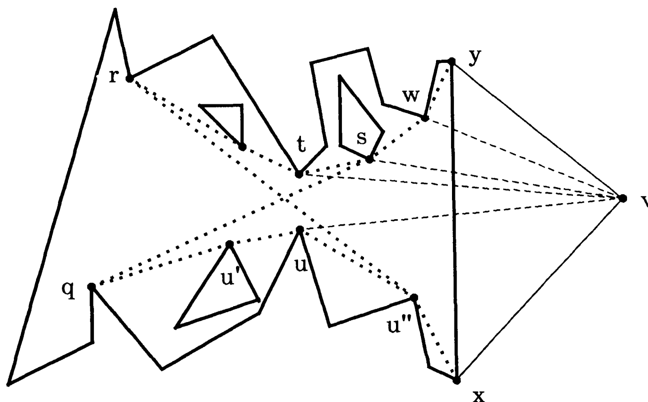


FIG. 10. *Searching for a new visible apex.*

(a) If u' is undefined, because the apex associated with (u, v) is a leaf in the lower tree of FNL (v, y), then it follows that there is no apex of FNL (x, y) whose visibility of v is blocked by u, and so we can take q to be u and r to be the successor of u in funnel order, and continue with step (2).

(b) Otherwise let us think of u' (represented by the segment from u' to u) as an apex in FNL (x, y). Let S denote the set of apexes who are descendents of either u' or of its siblings in the lower tree lying counterclockwise of u'. These apexes will be consecutive in FNL (x, y), starting from just after u to q. These are exactly the apexes whose visibility of v is blocked by u. The first apex r immediately following S in the funnel order of FNL (x, y) will be the postorder successor of u' (ignoring the ancestors of u') in a clockwise traversal of the lower tree of (x, y).

The apex r can be found by the following loop which walks along the lower tree of (x, y) towards the root. Let $u'' \leftarrow u$. Throughout the loop we will maintain the invariant that u'' is the parent of u'. While u' is the extreme clockwise child of u'', let $u' \leftarrow u''$ and let u'' be assigned its parent in the lower tree of (x, y). On exit of this loop, let r be the next clockwise sibling of u' in the lower tree. (Since $u \neq y$, such a successor will eventually be found.) Take q to be the predecessor of r in funnel order.

(This traversal toward the root of the lower tree of (x, y) is complicated technically by the fact that the tree has been altered during previous calls to SPLIT by the addition of visible segments from these vertices to v. However, the data structure described in § 7 has no difficulty ignoring these added segments and taking their clockwise neighbors instead.)

(2) If $q \neq u$, all the funnels following u up to q are known to be hidden from v, but can see the edge (v, y). Split the list FNL $[u, w]$ just after u and up to and including q, yielding the sublist of apexes a such that $u < a \leqq q$ (in funnel order). Concatenate this sublist to the end of FNL (v, y).

(3) Let s be the parent of q in the upper tree. The following properties relating r, s, and w are now relevant. These are proven later in Lemma 5.3.
   (a) Both w and s lie on the upper chain from r to y so that s lies between r and w (inclusive) on this chain.
   (b) The set of apexes on the upper chain from r to y that are visible from v form a contiguous subchain whose last element is either r or an apex t such that the line passing through v and t is tangent to the upper chain.
   (c) The segment (v, t) is the next visible segment after (v, u) in clockwise order about v.
   (d) The apex s is visible from v; that is, s lies between t and w on this upper chain.

Property (d) is key to the procedure since it implies that we have "jumped" from one visible vertex u to another visible vertex s in essentially constant time. The other properties are used to help locate the intermediate visible vertices.

The vertex that we are really interested in finding is the vertex t, which closes off the pocket started by u. Unfortunately, our search procedure only gives us s, a visible ancestor of t, in the upper tree. It would be tempting to simply search for t at this point, but in order to maintain our complexity bounds, we must make each piece of work pay off with the discovery of a new visible

segment. The remainder of the procedure "mops up" the pockets of visibility between t and w.

(4) Traverse the upper parent chain from s to w, and then backtrack along this chain from w back through s and towards r. (Backtracking is done by stacking the vertices visited from s to w and then popping the stack, while the traversal from s towards r is done by selecting the next clockwise segment in the upper tree following the segment (s, q) and then continuing along the extreme counterclockwise child of each succeeding apex. Since r is the next vertex in the upper tree following q in postorder, this process will eventually terminate at r if allowed to.) This traversal continues until reaching r or the last apex t that is visible from v. (The apex t is a point of tangency on the upper chain with respect to v (see Fig. 10).) From properties (3)(b) and (3)(d) above it follows that all of the apexes visited by these traversals are visible from v. Let $t_0, t_1, \cdots, t_k$ denote the apexes visited by this traversal in reverse order so that $t = t_0$ and $w = t_k$.

(5) The counterclockwise turning of the upper chains implies that every apex a, $r \leqq a < t$ in the funnel order, will have its visibility from v blocked by t, but each apex will be visible from the lower edge (x, v). If $r = t$, then this sublist is empty, otherwise split FNL $[u, w]$ just before r and just before t and concatenate this sublist to the end of FNL (x, v).

(6) By Lemma 5.2 the apexes $t = t_0, t_1, \cdots, t_k = w$ are given in clockwise order about v, are all visible from v, and in each case, t_i is the parent of t_{i-1} in the upper tree. It is easy to see that FNL $[u, w]$ consists of the funnels between u and t, which have already been processed, and the concatenation of FNL $[t_{i-1}, t_i]$ for $i = 1, 2, \cdots, k$ (including also the visible segments (v, t_i)). Thus the preconditions of the SPLIT procedure apply. For i running from 1 to k do the following.
   (a) Add the visible segment (v, t_{i-1}), thus effectively splitting the funnel with apex t_{i-1} into two funnels, a lower funnel whose base is edge (x, v) and an upper funnel whose base is edge (v, y).
   (b) Concatenate the lower funnel to the end of FNL (x, v) and concatenate the upper funnel to the end of FNL (v, y).
   (c) Call SPLIT (t_{i-1}, t_i). This will find all the visible apexes between t_{i-1} and t_i and will append all funnels to either FNL (x, v) or FNL (v, y) as appropriate.

The only nontrivial observations needed to establish the correctness of SPLIT are the properties mentioned in step (3).

LEMMA 5.3. *Properties* (a), (b), (c), *and* (d) *listed in step* (3) *of the above algorithm are all true.*

Proof. Since q and r are consecutive in funnel order, where q precedes r, we can apply Lemma 5.1 to the hourglass of q and r. As in that lemma, if r is the parent of q in the upper tree, then the hourglass degenerates into a funnel, and $s = r$ and is visible from v. The lemma follows immediately from basic funnel properties. It was shown in the proof of Lemma 5.1 that s lies between r and y on the upper chain. We show that s is between w and r. By Lemma 5.1 and the convexity of the upper and lower chains, since u is an ancestor of q on the lower chain, the parent of u, namely w, is an ancestor of the parent of q, namely s, on the upper chain. This establishes (3)(a). Property (3)(b) is a simple consequence of Lemma 5.1 and the convexity of the chains.

We argued earlier that all apexes strictly after u and up to q are hidden from v. To prove (3)(c) we first argue that all apexes starting with r, and up to but not including t, are also hidden from v. Property (3)(c) will then follow from Lemma 5.2, because t will be the next visible vertex in funnel order. If $r = t$, then this sequence of apexes is empty and the claim is trivially true. Otherwise the line passing through v and u is tangent to the lower chain from q to x (or possibly $q = u$). The apex r lies on the opposite side of this line because r's visibility of v is not blocked by u. Since $r \neq t$, t is a point of tangency with respect to v along the upper chain. By extending the segments (v, u) and (v, t) through u and t, respectively, to the boundary of P, we have a wedge that separates q from r. This wedge contains no apexes in its interior, for otherwise q and r would not be adjacent in funnel order. Since t is an ancestor of r in the upper tree, all the successors of r up to, but not including, t are descendents of t in the upper tree (because funnel order corresponds to a postorder traversal of the upper tree), and it is easy to see that t is a point of tangency with respect to v for the upper chains of all of these successors. Thus all these apexes are hidden from v, implying that t is the next visible apex.

To prove property (3)(d) we claim that s lies on the portion of the upper chain from r to y which is visible from v. Since this chain is convex and t is a point of tangency, this means that we must show that s lies on the portion of this upper chain from t to y. Suppose that s were to lie in the invisible portion of the upper chain from r to t. Consider the upper tree edge from q to s. Because this segment is tangent to the upper chain from r to y (and is directed so that its extension through s would stab the segment (x, y)) it would follow that q lies clockwise from t with respect to v. However, by our construction, q's visibility of v is blocked by u (or q equals u), so q lies counterclockwise of u with respect to v. This leads to a contradiction because we have just shown that t is clockwise of u with respect to v. □

Ignoring the time needed to manipulate the underlying data structure (which we will show to be $O(E)$ in § 7), the algorithm's running time is proportional to the number of visible segments added.

LEMMA 5.4. *Assuming that the graph is represented as an enhanced visibility graph, the running time of* SPLIT (x, y) *is proportional to the number of visible segments added to* v.

Proof. The procedure performs essentially only local traversals of the enhanced visibility graph, by walking around either the lower or upper trees for the edge (x, y). As mentioned in Lemma 4.1, these traversals can be performed in constant time assuming that the graph is represented as an enhanced visibility graph.

Let E_v denote the number of visible segments added to v during the call SPLIT (x, y). To show that SPLIT runs in $O(E_v)$ time note that the first argument to SPLIT is always an apex visible from v, and since successive calls are to apexes in further funnel order, SPLIT is never called with this same first argument twice. Thus, the number of recursive calls is at most E_v. The procedure contains only two loops. The first loop appears in step (1)(b) when the lower chain is searched starting from u for the vertex u'' that is the parent of r in the lower tree. Each apex visited in this loop is visible, and we claim that, with the exception of the apex u'', none of these apexes will be visited twice by this loop. The reason is that all subsequent executions of this loop will begin searching starting from some apex that comes after (or is equal to) the apex t in funnel order. Since t is an ancestor of r on the upper chain, t's ancestors on the lower chain will be ancestors of the parent of r, namely u''.

The second loop appears in step (4) where the upper chain from s back to w is traversed and then retraversed to t. All of the apexes visited in this process are visible

by Lemma 5.3, and a recursive call is made for each such apex (except w), and so the cost of this step cannot exceed $O(E_v)$ altogether. □

6. The overall algorithm. Finally we describe how to use the SPLIT procedure to compute the enhanced visibility graph for a polygonal domain P. The problem reduces to that of incorporating a new vertex v into a triangulated region P resulting in an enlarged triangulated region P'. The difference between this process and the problem that SPLIT solves is that SPLIT incorporates exactly one new triangle into P, and in the plane-sweep triangulation we incorporate one or two sequences of triangles whose bases form an inward-convex chain with respect to v by Lemma 2.1.

For each of these sequences of triangles, let $u_0, u_1, \cdots, u_m$, be vertices on the inward-convex chain that are visible from v. We consider three cases.

(1) If the sequence is empty, v cannot see any vertex on P, implying that there is no change in the visibility graph except the inclusion of the isolated vertex v. This occurs in the plane sweep whenever a vertex is inserted whose local neighborhood with respect to P lies to the right of the vertical line passing through v.

(2) If the sequence contains one vertex u_0, then v can see only u_0 on P (locally), implying that v cannot see any other vertices within P. Thus the only change in the visibility graph is the inclusion of the edge (v, u_0). This will be the case, for example, for a vertex following case (1). We will think of this single edge as consisting of two oppositely directed edges that bound a polygon with zero area. The only funnels are degenerate funnels.

(3) Otherwise, the sequence contains at least two vertices forming an inward-convex chain with respect to v. In the rest of the discussion, we consider this case.

Each triple (u_{j-1}, u_j, v) forms a triangle (see Fig. 11). The edges (u_0, v) and (v, u_m) are on the boundary of P'. Our objective is to compute FNL (u_0, v) and FNL (v, u_m) (for P'). Assume inductively that we have already computed FNL (u_{j-1}, u_j) for P for each of the edges (u_{j-1}, u_j) on the chain (since this will be a part of the representation of the enhanced visibility graph for P). For each such edge and vertex v, call the SPLIT procedure. This splits FNL (u_{j-1}, u_j) into two funnel sequences, one for the lower edge (u_{j-1}, v), which we call L_j, and the other for the upper edge (v, u_j), which we call U_j. In the process, SPLIT also adds all the visible segments from v passing through the edge (u_{j-1}, u_j). Although L_j consists of funnels for the polygon P whose common base is the edge (u_{j-1}, v), we can think of them as funnels for P' whose

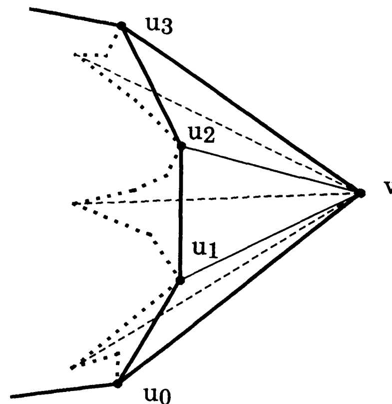


FIG. 11. *Joining a vertex to an inward-convex chain.*

common base is the edge (u_0, v). Similarly, U_j can be thought of as a sequence of funnels for P'. In order to form the desired funnel sequences for P' we appeal to the following lemma, which establishes that we can obtain this funnel sequence by concatenating the intermediate sequences.

LEMMA 6.1. *Consider a point v external to a bounded polygonal domain P and an inward-convex chain of vertices $u_0, u_1, \cdots, u_m$ $(m \geqq 1)$ on the boundary of P visible from v. Let P' be the polygon obtained by replacing this chain with the edges (u_0, v) and (v, u_m). Then*

(i) *FNL (u_0, v) in P' is equal to the concatenation of L_j for $1 \leqq j \leqq m$, followed by the trivial funnel whose apex is v.*

(ii) *FNL (v, u_m) of P' is equal to the concatenation of the trivial funnel whose apex is v followed by U_j for $1 \leqq j \leqq m$.*

(iii) *The computation of L_j and U_j does not affect the computation of L_i and U_i for any $i \neq j$.*

Proof. We first prove (i), and (ii) follows by a symmetric argument (together with Lemma 3.2). Consider the lower tree for FNL (u_0, v). Each vertex u_j is visible from v and hence is the apex of a funnel for edge (u_0, v) whose lower chain consists of $u_0, u_1, \cdots, u_j$ and whose upper chain consists of the single segment (u_j, v). To see that this forms a funnel, observe that the chain $u_0, \cdots, u_j$ is inward-convex with respect to v, and (u_0, v) is an edge of P'. In general, if C is a path in the lower tree for edge (u_{j-1}, u_j) in P, then the concatenation of $u_0, u_1, \cdots, u_{j-1}$ with C forms a chain in the lower tree for edge (u_0, v) in P'. Conversely, other than the segment (u_0, v) (corresponding to a trivial funnel), every path in the lower tree for edge (u_0, v) is of the form $u_0, u_1, \cdots, u_{j-1}$ followed by some chain C in the lower tree for edge (u_{j-1}, u_j). Hence, every funnel in L_j is extendible to a funnel of (u_0, v) and the funnel order within L_j is preserved in the extension. Since u_{j-1} is the child of u_j in the lower tree for (u_0, v), L_{j-1} precedes L_j in the funnel order for (u_0, v). This implies that FNL (u_0, v) is the concatenation of FNL (u_{j-1}, u_j) (which is L_j) for $1 \leqq j \leqq m$, followed by v.

To prove (iii) we first note that there are two things that might go wrong. First, when calling SPLIT, the list FNL (u_{j-1}, u_j) is destroyed to form L_j and R_j. However, SPLIT does not access any funnel sequences other than the one that it is decomposing, and since funnel sequences are disjoint, one decomposition does not affect another one. The second thing that may occur is that SPLIT adds visible segments from v to some of the vertices in P in order to form the visibility graph for P'. It might be that by adding these segments, we would alter the structure of a funnel in some other funnel sequence (since we use the visibility structure to traverse the funnel trees). Observe that it may be the case that this visible segment intersects the interior of a funnel for some other base edge, or even for this base. The key is that this visible segment does not alter the set of funnels or the funnel structure for any other base edge. To see this, consider the apex for a funnel located at some vertex v belonging to some other base edge e. As shown earlier, the first visible segments of the upper chain and lower chain for this apex define a wedge whose apex is v such that all rays emanating from v intersect the boundary of P first at the base e. The only way that the newly added segment could affect the structure of this funnel would be if the newly added visible segment is incident upon v and lies within this wedge. However, the fact that the new visible segment first intersects the base edge (u_{j-1}, u_j) implies that it cannot lie within this wedge. By applying this argument to every funnel of the lower tree for base edge e, we see that the newly added visible segment cannot alter the lower tree for e, and hence it cannot alter the funnel structure for e. □

Thus, the overall algorithm for building the enhanced visibility graph of P follows.

(1) Compute the plane-sweep triangulation of P by forming the triangulated polygons with holes $P_1, P_2, \cdots, P_n$. The enhanced visibility graph of P_0 is empty. For k running from 1 to n, repeat steps (2) through (4).
(2) When v_k is added to the triangulation it is connected to either one or two inward-convex chains of vertices on the boundary of P_{k-1}. For each such chain $u_0, u_1, \cdots, u_m$, perform steps (3) through (4).
(3) If the chain has length zero, then simply add the isolated vertex v_k to P_{k-1} forming P_k. If the chain has length one, then add the vertex v_k and the visible segment (v, u_0) to P_{k-1}, forming P_k.
(4) If the chain has length two or greater, call SPLIT on the polygon P_{k-1} with each edge (u_{j-1}, u_j) and vertex v_k (for $1 \leqq j \leqq m$) forming L_j and U_j. Concatenate the L_j's together with the trivial funnel whose apex is v_k and whose base is (u_0, v_k) to form FNL(u_0, v_k). Concatenate the trivial funnel whose apex is v_k and whose base is (v_k, u_m) together with the U_j's to form FNL(v_k, u_m). From these we have the enhanced visibility graph for P_k.

The running time of the complete visibility graph algorithm is proportional to the sum of the times to:

• compute the plane-sweep triangulation, which we showed to be $O(n \log n)$, plus the number of edges in the triangulation, which is $O(n)$; and

• the time needed to call the procedure SPLIT for each triangle of the triangulation, which we will show to be $O(E)$ in the next section (where E is the number of visible segments in the visibility graph).

7. Data structure. The only detail omitted in the previous sections is how the operations REV, CW, CCW, CX, and CCX are implemented. An earlier version of this paper used finger trees to implement these operations [4]. In this version we use a simpler data structure based on a data structure for the set Split-Find problem [3].

To implement the operations of CW and CCW, all that is needed is a doubly linked adjacency list for each vertex such that the entries are sorted in angular order about each vertex. To implement REV, we cross index entries for oppositely directed edges. Note that when inserting new visible segments in the SPLIT procedure, we always have access to the clockwise neighbors of the new segment (because the segments are always inserted into the middle of a funnel apex, and each apex is represented by the first segment of the lower chain). Thus updates can be performed in $O(1)$ time.

To compute the boundary extensions CX and CCX we will need to make use of the following observations about the way in which visible segments are added to this structure. Every vertex v has two phases during which segments are added to it. Phase A occurs when we are incorporating the new vertex v into the visibility graph in the SPLIT procedure. All the visible neighbors of v discovered during this phase have already been visited (have lower x-coordinates) and have already been incorporated into the visibility graph. As observed in the earlier sections, the visible neighbors added during this phase are added in clockwise order about v. Phase B neighbors arise when a vertex u whose x-coordinate is higher than v's is being incorporated into the visibility graph, and the SPLIT procedure applied to u discovered that some funnel whose apex is at v can see u. We have no control over the order (about v) in which these neighbors appear. All phase A neighbors have been added before any phase B neighbors are added.

A vertical line passing through v divides the plane into two halfplanes; the left halfplane contains the phase A visible neighbors and the right halfplane contains the phase B visible neighbors. We will maintain the segments of each phase in clockwise angular order about v, and we make the convention that there are imaginary vertical

segments, so that each segment has a predecessor in this order. Extend each phase B visible segment to a line passing through v. These linear extensions subdivide the left halfplane into a set of wedges about v. These wedges divide the phase A segments into disjoint intervals of segments. Each interval is associated with the phase B segment whose extension is the nearest counterclockwise extension to (immediately preceding) the interval, and each phase B segment is associated with the first nonempty interval that lies clockwise from (immediately after) its linear extension (see Fig. 12).

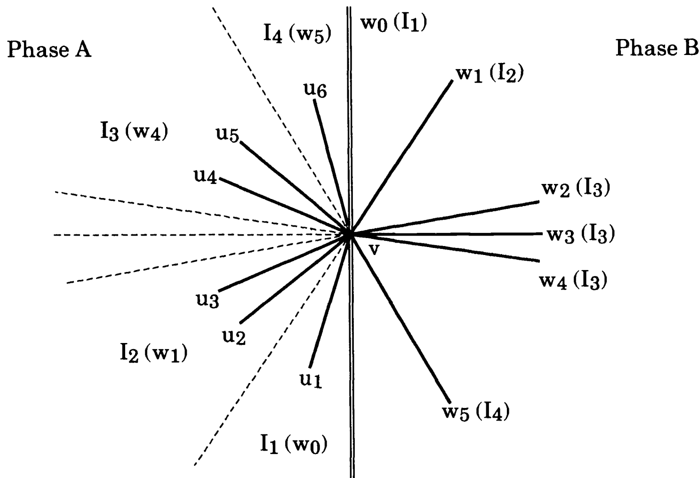


FIG. 12. *Visible neighborhood of a vertex.*

By maintaining this interval partition of the phase A neighbors, we claim that we can compute the extensions CX and CCX. Define the clockwise extension *candidate* of segment (u, v) to be the visible segment with the smallest clockwise angle greater than 180 degrees with respect to (u, v). The candidate differs from the true clockwise extension in that the clockwise extension may not be defined if one of the two boundary edges of P intersects v locally through this angular sweep. Clearly, it can be tested in constant time whether the clockwise extension candidate is the true clockwise extension. A similar definition applies to the counterclockwise extension candidate. If (u, v) is a phase A segment, then its counterclockwise extension candidate is the extension edge associated with the interval containing this segment, and the clockwise extension candidate is the clockwise neighbor of this segment. If (w, v) is a phase B segment, then the clockwise extension candidate for this edge is the first segment in the interval associated with this segment and the counterclockwise extension candidate is the last segment in the previous interval.

From this, it is clear that maintaining extensions can be reduced essentially to the problem of maintaining a partition of the phase A visible segments of v into a set of intervals, where the intervals are defined by the linear extensions of phase B visible segments of v. Let m_A denote the number of v's visible neighbors in A. Since the phase A neighbors are added in clockwise order, we can easily associate them with the set of integers $S = \{1, 2, \cdots, m_A\}$. Observe that this is just a ranking of the visible segments in order of decreasing slope. This is done on completion of the SPLIT procedure when v is incorporated into the visibility graph.

These integers are stored in a data structure developed by Gabow and Tarjan for processing Split-Find operations [3]. The Split-Find data structure (not to be confused

with the procedure SPLIT) is designed to process an intermixed sequence of the following two operations, which are seen to be a reversal of the familiar Union-Find operations:

Find(i): Return the name of the set containing i.
Split(i): Split the set containing i into two sets, one containing all integers less than or equal to i, and the other containing all integers greater than i.

Given an initial set of size a, a sequence of b Splits and Finds can be processed in total time $O(a+b)$. In addition, each Find runs in constant time.

As mentioned earlier, the Split-Find data structure is initialized to contain the integers S associated with the phase A visible segments as soon as the SPLIT procedure has completed. Before describing the processing of the phase B visible segments, there is one operation which we will need to discuss which is not supported directly by the Gabow and Tarjan data structure. When a new phase B segment is discovered, we need to find the counterclockwise extension candidate, that is, the next larger phase A segment in slope order. Note that this is not the same as a Find operation because Find assumes that the exact index of the split point is known. We assume that the slopes of the phase A segments are stored in an array sorted by decreasing slope. This order is available to us without sorting because the visible segments are added in slope order.

To update the structure when a new phase B visible segment (v, w) is added, recall that we know the existing phase B segment, say (v, w'), immediately preceding this segment in clockwise order about v. The phase A interval I associated with w' will be the interval split by the extension of the new segment. To locate the counterclockwise extension of (v, w), we perform a *dovetailed doubling search* starting at each end of the interval I. This is done by locating the endpoints of the interval I in the slope array, and performing two one-sided doubling searches starting in from opposite ends of the interval, dovetailing the operations of the two searches into an interleaved sequence. (Observe that this is essentially a simple implementation of the search performed by finger search trees.) It follows that the time required to locate the clockwise extension is proportional to the logarithm of the distance to the nearer of the ends of the interval.

If the counterclockwise extension of (v, w) is before the first segment of I, then we associate (v, w) with I and do not split the interval. If (v, w) is the extension immediately preceding I, then we update I's associated phase B segment. If the counterclockwise extension of (v, w) is the last segment in I, we associate (v, w) with the successor interval of I. Each such trivial search requires constant time, so overall their running times are bounded by $O(m_B)$, where m_B is the number of phase B visible segments incident upon v. Otherwise, we apply the dovetailed search procedure described above to locate the counterclockwise extension of (v, w). Let us say that the index of this extension is i. We call Split(i), associating the new interval of elements that are greater than i (clockwise from u_i) with (v, w). It follows that when applied to an interval of size m_A, the asymptotic running time of this algorithm satisfies the recurrence

$$T(m_A) = \max_{1<k<m-1} T(k) + T(m_A - k) + \min(\log k, \log(m_A - k)),$$

whose solution is $O(m_A)$ (see [12, p. 185]). Combining this with the $O(m_B)$ cost for the trivial finds implies that the total time spent searching for extensions about the vertex v is $O(m_A + m_B)$. Summed over all vertices, the total running time of the searches is $O(E)$.

Each CX and CCX operation performs one Find operation (observe that no Find is needed on a phase B segment, since we simply access the first or last segment in the appropriate interval). Thus each CX or CCX operation requires $O(1)$ time in our data structure, and hence $O(E)$ time overall, since our algorithm performs this many primitive operations. Each Split arises when a visible segment is added in phase B, of which there are at most m_B. Thus the total amount of time spent in the Gabow and Tarjan data structure processing the Splits is $O(m_A + m_B)$, which again is $O(E)$ when summed over all vertices.

8. Concluding remarks. We have given an $O(n \log n + E)$ algorithm for constructing the visibility graph of a set of polygonal obstacles in the plane. The construction is based on the notion of funnels, funnel sequences, and upper and lower trees, which have arisen in various forms in the study of visibility and shortest paths in polygons. These notions are combined with a novel method of traversing the visibility graph utilized in the procedure SPLIT. Together with a variation of Dijkstra's algorithms that runs in $O(n \log n + E)$ time, this provides a shortest-path algorithm in the midst of polygonal obstacles whose running time is dependent on the size of the visibility graph.

The principal drawback of our algorithm is the complexity of its implementation, particularly due to the extraction of the tree traversal primitives from the enhanced visibility graph. As an implementation note, there is a simpler data structure for the Split-Find problem that runs in $O(m \log^* n)$ time [7]. Although this leads to a theoretically slower algorithm, $O(n \log n + E \log^* n)$, it is likely that the simpler version will run faster for all reasonable input sizes. Another interesting issue is that the algorithm may need to store $O(E)$ segments at every intermediate stage. Overmars and Welzl's $O(E \log n)$ visibility graph algorithm, although inferior with respect to asymptotic complexity, requires only $O(n)$ working storage [14]. The need to store the complete visibility graph at every stage of the algorithm seems inherent in our approach.

Other sorts of visibility graphs are easily derivable from this algorithm. It is a fairly simple enhancement to the algorithm to label each funnel apex with the unique edge that can be seen by looking out from the apex through the funnel. From this the vertex-edge weak visibility graph can be derived (where a vertex and edge are adjacent if the vertex can see at least one point of the edge). The visibility polygon of a vertex can be constructed in $O(n)$ time. The edge-edge weak visibility graph can also be derived (where two edges are adjacent if they contain points which are mutually visible) since two edges e_1 and e_2 are weakly visible if and only if there exist vertices u and v such that the funnel apex whose lower chain begins with the segment (u, v) sees edge e_1, and the funnel apex whose lower chain begins with edge (v, u) sees edge e_2. Although the running time of the algorithm is dependent on the size of the standard visibility graph E, and not on the size of the edge-edge weak visibility graph E_W, it can be shown that these two quantities are asymptotically equal. To see this, observe that the above construction implies that $E_W \in O(E)$. Any pair of visible vertices (u, v) can be associated with a weak visibility between two edges of P having u and v as endpoints. Each weakly visible pair of edges is associated with at most four such visible pairs, and so $E \in O(E_W)$ (we thank one of the anonymous referees for this observation).

In general, not all of the visibility graph is needed by the shortest-path algorithm. In general, shortest paths will travel only along the lines of tangency between the obstacles. Kapoor and Maheshwari [8] have shown that such a reduced visibility graph can be computed in $O(E_R + T)$ time, where T is the time needed to triangulate the polygonal domain, and E_R is the number of edges in the reduced visibility graph.

Acknowledgments. The authors would like to thank John Hershberger and Subhash Suri for observing that the weak visibility graphs can be derived from this algorithm, and Kurt Mehlhorn for pointing out the connection between our plane-sweep triangulation and the plane-sweep triangulation algorithm for simple polygons. We would also like to thank Joe Mitchell for his careful reading of an earlier draft of this paper. Finally, we would like to thank the anonymous referees for their careful reading and valuable suggestions, and in particular for finding a subtle error in the application of the Gabow and Tarjan data structure.

REFERENCES

[1] T. ASANO, T. ASANO, L. GUIBAS, J. HERSHBERGER, AND H. IMAI, *Visibility of disjoint polygons*, Algorithmica, 1 (1986), pp. 49-63.

[2] 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.

[3] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209-221.

[4] S. K. GHOSH AND D. M. MOUNT, *An output sensitive algorithm for computing visibility graphs*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 11-19.

[5] L. GUIBAS, J. HERSHBERGER, D. LEVEN, M. SHARIR, AND R. E. TARJAN, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209-233.

[6] J. HERSHBERGER, *An optimal visibility graph algorithm for triangulated simple polygons*, Algorithmica, 4 (1989), pp. 141-155.

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

[8] S. KAPOOR AND S. N. MAHESHWARI, *Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles*, in Proc. 4th ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 172-182.

[9] D. T. LEE, *Proximity and reachability in the plane*, Ph.D. thesis and Tech. Report ACT-12, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.

[10] D. T. LEE AND F. P. PREPARATA, *Euclidean shortest paths in the presence of rectilinear boundaries*, Networks, 14 (1984), pp. 393-410.

[11] T. LOZANO-PEREZ AND M. A. WESLEY, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. ACM, 22 (1979), pp. 560-570.

[12] K. MEHLHORN, *Data Structures and Algorithms, Volume* 1: *Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[13] K. MEHLHORN, *Data Structures and Algorithms, Volume* 3: *Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[14] M. H. OVERMARS AND E. WELZL, *New methods for constructing visibility graphs*, in Proc. 4th ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 164-171.

[15] M. SHARIR AND A. SCHORR, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193-215.

[16] E. WELZL, *Constructing the visibility graph for n line segments in* $O(n^2)$ *time*, Inform. Process. Lett., 20 (1985), pp. 167-171.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 911-935, October 1991

© 1991 Society for Industrial and Applied Mathematics
007

LEARNING SIMPLE CONCEPTS UNDER SIMPLE DISTRIBUTIONS*

MING LI† AND PAUL M. B. VITÁNYI‡

Abstract. This paper aims at developing a learning theory where "simple" concepts are easily learnable. In Valiant's learning model, many concepts turn out to be too hard (like NP hard) to learn. Relatively few concept classes were shown to be learnable polynomially. In daily life, it seems that things we care to learn are usually learnable. To model the intuitive notion of learning more closely, it is not required that the learning algorithm learn (polynomially) under all distributions, but only under all simple distributions. A distribution is simple if it is dominated by an enumerable distribution. All distributions with computable parameters that are used in statistics are simple. Simple distributions are complete in the sense that a concept class is learnable under all simple distributions if and only if it is learnable under a fixed "universal" simple distribution. This holds both for polynomial learning in the discrete case (under a modified model), and for non-time-restricted learning in the continuous case (under the usual model). This completeness result is used to obtain new learning algorithms and several quite general new learnable classes. These include a discrete class that is known to be not polynomial learnable under Valiant's model, unless RP = NP, and a continuous class that is not learnable in Valiant's model. The results here allow that for each concept class from a wide range of concept classes, for each underlying distribution from a wide range of distributions, the learning algorithm uses a single fixed procedure to draw examples by a single algorithmic process using a random number generator. The "universal" simple distribution is not computable. To make the theory feasible, a polynomial-time version is developed for it. All results derived for discrete sample spaces hold *mutatis mutandis* for the polynomial-time versions, including versions of completeness, the new learning algorithms, and the new learnable classes.

Key words. PAC learning, Kolmogorov complexity, universal distribution, enumerable distributions, learning simple concepts, completeness, discrete and continuous sample spaces, polynomial-time learning algorithms

AMS(MOS) subject classifications. 68G05, 68C05, 68C25

1. Introduction. Valiant has proposed a learning theory in which one wants to learn a concept with high probability, in polynomial time, and a polynomial number of examples, within a certain error, under all distributions on the examples [V]. A precise definition of this "pac-learning" is given in § 1.2. Let us highlight its special features. It contrasts with the common approach in statistical inference, or recursion theoretical learning, where we want to learn a concept precisely in the limit, by insisting only on learning a concept approximately. The feasibility restriction to a polynomial algorithm precludes the precise learning of nontrivial concepts, and therefore we had to relax precision to within a certain error. This corresponds with natural learning, where it is important to learn fast, and it suffices to learn approximately. The additional computational requirements are orthogonal to the usual concerns in inference, and result in a distinct novel theory. But many subsequent investigations have demonstrated negative, hardness, or equivalence results [G], [A2], [KLPV], [PW], [KV], [PW1], [PV]. There are at least two problems with Valiant's proposal in [V]:

* Received by the editors April 25, 1990; accepted for publication (in revised form) December 6, 1990. An extended abstract of this paper appears in the Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 34–39.

† Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. The work of this author was supported by Natural Sciences and Engineering Research Council of Canada operating grants OGP-0036747 and OGP-046506. He performed part of this work while at the Computer Science Department, York University.

‡ Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, the Netherlands and Universiteit van Amsterdam, Faculteit Wiskunde en Informatica, Amsterdam, the Netherlands. The work of this author was partially supported by Natural Sciences and Engineering Research Council of Canada International Scientific Exchange Award ISE0046203.

(1) Under all distributions, many concept classes, including some seemingly simple ones, are not known to be polynomially learnable or known not to be polynomially learnable if RP$\neq$NP, although some such concept classes are polynomially learnable under some fixed distribution.

(2) In certain situations it may be undesirable, too slow, or impossible to sample according to underlying distributions. We aim at developing a theory in which a learning program, for *any* concept class it can learn, under *any* distribution from a large class of distributions, draws the examples using a single fixed table, representing a particular distribution, and a random number generator. This contrasts having to go out in the real world and draw its examples from the actual underlying distribution.

Item (1) is at odds with the notion that machine learning should be practically useful. One may interpret it as evidence that Valiant's initially proposed requirements for learning are too strong. In practice, it seems exaggerated to require that the algorithm learn under *all* distributions. Accordingly, several authors have proposed to study Valiant learning under particular distributions [KLPV], [N], [BI]. Then some previously (polynomially) unlearnable classes become learnable. For instance, the class of μDNF formulae is polynomially learnable under the uniform distribution. However, the assumption of *any special* distribution is obviously too restrictive and not practically interesting. There arises the problem of finding a class of distributions which is small enough to improve learning ability, but still large enough to be meaningful.

1.1. A new approach. We propose to study Valiant-style learning under *all simple distributions*, which properly include *all* computable distributions. This allows us to systematically develop a theory of learning for *simple concepts* that intuitively should be polynomially learnable. To stress this point: maybe it is too much to ask to be able to learn all finite automata fast, but surely we ought to be able to learn a *sufficiently simple* finite automaton fast. Previous approaches looked at syntactically described classes of concepts. We introduce the idea of restricting a syntactically described class of concepts to the concepts that are simple in the sense of having low Kolmogorov complexity. This will cover most intuitive notions of simplicity. Our other restriction, from distribution-independent learning to simple-distribution-independent learning is also not much of a restriction. *All* distributions we have a name for, like the uniform distribution, normal distribution, geometric distribution, and Poisson distribution, are recursive or enumerable—if we use finite precision parameters.

In many situations sampling according to the real distribution, as prescribed in Valiant's model, is problematic. In practice the examples are more conveniently provided algorithmically, rather than by drawing them from the underlying distribution.

Benevolent teachers provide good examples first in order to train a pupil fast. To learn addition, the teacher starts with "1+1" rather than with "592+4124." Providing the simpler examples first intuitively helps to improve the speed of learning. The results in this paper supply evidence for this thesis.

Consider a situation where a robot wants to learn but there is nobody around to provide it with examples according to the real distribution. Because it does not know the real distribution, the robot just has to generate its own examples according to its own (computable) distribution and do experiments to classify these examples (see [RS]). An example is the case of learning a finite state black box (with resetting mechanism and observable accepting/rejecting behavior).

If we want to put a man on the moon we cannot learn according to the real distribution. This is too expensive. Learning to drive without a teacher in Boston is too dangerous. We learn according to easily describable emergencies.

A solution is, that in *each case* the examples are algorithmically generated by the same program, whatever concept class or underlying distribution one is dealing with. This magical solution turns out to be realizable for a quite general range of concept classes and distributions.

1.2. Basics of learning. We review some standard definitions and facts from pac-learning theory.

(1) Let X be a set, the *sample space.* A *concept* is a subset of X. A *concept class* is a set $C \subseteq 2^X$ of concepts. An *example of a concept* $c \in C$ is a pair (x, b) where $b = 1$ if $x \in c$ and $b = 0$ otherwise. A *sample* is a set of examples. We enumerate a concept class C as $c_1, c_2, \cdots$ by enumerating finite binary strings $s(c_1), s(c_2), \cdots$ representing the concepts. For instance, $s(c)$ is the binary encoding of a finite automaton, and c is the language accepted by that automaton.

(2) Let $c \in C$ be the target concept and P be a distribution on X. Given *accuracy parameter* ε, and *confidence parameter* δ, a *learning algorithm* A draws a sample S of size $m_A(\varepsilon, \delta)$ according to P, and produces a *hypothesis* $h = h_A(S) \in C$.

(3) We say C is *learnable by* C if there is a learning algorithm A, such that for all ε, δ, for every distribution P and every target concept $c \in C$, as in definition (2),

$$\mathbf{P}\{P(h \Delta c) > \varepsilon\} \leqq \delta,$$

where Δ denotes the symmetric difference between two sets, and $\mathbf{P}$\{boolean\} is the probability of boolean being true. In this case we say that C is (ε, δ)*-learnable*, or *pac-learnable* (*probably approximately correct*).

(4) C is *polynomially learnable* if A (ε, δ)-learns C, and runs in time polynomial (and asks for a polynomial number of examples) in $1/\delta$, $1/\varepsilon$, and the length of the representation $s(c)$ of the concept c to be learned.

(5) We also consider the case where the Learning Algorithm A returns $h \in C'$ satisfying definition (3), rather than $h \in C$. In this case, we say C is *learnable by* C', or simply, C is *learnable.*

Remark. A different model as used by [V], [KLPV] assumes separate distributions over positive and negative examples. These models are basically equivalent. Also see [HLW] for an on-line model.

We need the following very useful theorem proved by Blumer, Ehrenfeucht, Haussler, and Warmuth [BEHW]. See also [KL] for the case when the concept is only consistent with a fraction of the examples.

OCCAM'S RAZOR THEOREM. *Let C and C' be concept classes. Let $c \in C$ be the target concept, and let n be the length of its binary representation $s(c)$. Let A be an algorithm which (ε, δ)-learns C. Let $\alpha \geqq 1$ and $0 \leqq \beta < 1$. Assume that A, using a sample S of m (positive and negative) examples drawn randomly from a distribution over the sample space, output a hypothesis $h \in C'$, which is consistent with at least $(1-\varepsilon/2)m$ examples in S, and its representation $s(h)$ has binary length less than or equal to $n^\alpha m^\beta$. If*

$$m = \Omega\left(\max\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}, \left(\frac{n^\alpha}{\varepsilon}\right)^{1/(1-\beta)}\right)\right),$$

then A learns C polynomially. If $\beta = 0$, and $n > \log(1/\delta)$, then we use $m = O(n^\alpha/\varepsilon)$. In many cases, we have $C = C'$.

1.3. Outline of the paper. We treat both discrete and continuous concept learning.

While each discrete concept class is learnable in unrestricted time, this is not the case for concept classes over continuous sample spaces. In the paper, for expository

reasons, we first treat the discrete case, and then the continuous case. To outline the results, the reverse order seems more convenient.

In § 4, we derive a completeness result for continuous concept learning. There exists a "universal" measure such that a concept class is learnable under this *single* measure if and only if it is learnable under *all* "simple" measures. This result holds both if we sample according to the actual simple measure itself, or according to the substitute "universal" measure. Subsequently, we use the result to show that there is a concept class which is learnable under all simple measures, but which is not learnable under all measures.

In § 2 we derive a completeness result for polynomial learning of discrete concepts. There exists a "universal" distribution such that a concept class is polynomially learnable under this *single* distribution if and only if it is polynomially learnable under *all* "simple" distributions, provided we sample according to the "universal" distribution. We use this completeness result as a novel tool to obtain new nontrivial learning algorithms for several (old and new) classes of concepts in our model. For example, the class of DNFs such that each monomial has Kolmogorov complexity $O(\log n)$, the class of simple DNF, the class of simple k-reversible DFA (in the Appendix), and the class of monotone k-term DNF, are polynomially learnable under our assumptions. These classes are not known to be polynomially learnable under Valiant's more general assumptions; monotone k-term DNF is not polynomially learnable in Valiant's model, unless RP = NP. We have put the treatment of simple k-reversible DFA in the Appendix, because the other examples already illustrate the point we want to make well enough.

The "universal" distribution in § 2 is not computable. In § 3 we develop the theory of § 2 for polynomial-time computable distributions. It turns out that this class also has a "universal" distribution, which is exponential-time computable. Apart from this, all results derived in § 2 hold *mutatis mutandis* in the polynomial-time setting, including the new learning algorithms and new learnable problems. We give some ideas about how to use the developed theory.

2. Discrete sample spaces.

NOTATION. Let N, Q, and R denote the set of nonnegative integers, nonnegative rational numbers, and nonnegative real numbers, respectively. A superscript "+," as in N^+, restricts the set involved to the positive numbers. If x is a binary sequence, then its *length* $l(x)$ is the number of occurrences of zeros and ones in it; if x is an integer, then $l(x)$ denotes the length of the binary representation of x.

DEFINITION. We consider countably infinite sample spaces, say $S = N \cup \{u\}$, where u is an "undefined" element not in N. A function P from S into R, such that $\sum_{x \in S} P(x) = 1$ defines a *probability distribution* on S. (This allows us to consider defective probability distributions on the natural numbers, which sum to less than one, by concentrating the surplus probability on u.) The function P itself is properly called the "probability density function," but we identify it loosely with the "probability distribution." A probability distribution P is called *enumerable*, if the set of points

$$\{(x, y) : x \in N, y \in Q, P(x) > y\},$$

is recursively enumerable. That is, $P(x)$ can be approximated from below by a Turing machine, for all $x \in N$. ($P(u)$ can be approximated from above.) Note that enumerable distributions include the recursive ones.

L. A. Levin has shown that we can effectively enumerate all enumerable probability distributions, $P_1, P_2, \cdots$. In particular, it can be proved that there exists a *universal enumerable probability distribution*, denoted by say $\mathbf{m}$, such that

$$\forall i \in N^+ \exists c > 0 \forall x \in N [c\mathbf{m}(x) \geqq P_i(x)]. \quad (1)$$

That is, $\mathbf{m}$ dominates each P_i multiplicatively. Let $K(x)$ be the prefix variant of Kolmogorov complexity first proposed by Levin [L], [G1]. It is defined as follows. Consider an enumeration $T_1, T_2, \cdots$ of Turing machines with a separate binary one-way input tape. Let T be such a machine. If T halts with output x, then T has scanned a finite initial segment of the input, say p and we define $T(p) = x$. The set of such p for which T halts is a prefix code, no such input is a proper prefix of another one. Fix a universal machine in this enumeration, say U, and call it the *reference* prefix Turing machine. Define

$$K(x) = \min \{l(p): U(p) = x\}.$$

It can be proved that, if $T(q) = x$ for some T in the enumeration and some program q, then $K(x) \leqq l(q) + c_T$, where c_T is a constant depending on T but not on x. It can be proved that if $T = T_i$, then $c_T = K(i) + O(1) \leqq \log i + 2 \operatorname{loglog} i + O(1)$. It can be proved that

$$\mathbf{m}(x) = 2^{-K(x)+O(1)}. \tag{2}$$

It can be proved that, in (1), the constant c can be set to

$$c = 2^{K(P_i)+O(1)} = 2^{K(i)+O(1)} = O(i \log^2 i). \tag{3}$$

This means that we can take c to be exponential in the length of the shortest self-delimiting binary program to compute P_i.

The universal distribution (rather, its continuous version) was originally discovered by Solomonoff in 1964, with the aim of predicting continuations of each finite prefix of infinite binary sequences [So]. We can view the discrete probability density $\mathbf{m}$ as the a priori probability of finite objects in the absence of any knowledge about them. Solomonoff's approach is as follows.

Consider the enumeration $T_1, T_2, \cdots$ of prefix Turing machines again. Assume the input is provided by tosses of a fair coin. The probability that T halts with output x is $P_T(x) = \sum_{T(p)=x} 2^{-l(p)}$, where $l(p)$ denotes the length of p. Then $\sum_{x \in N} P_T(x) \leqq 1$, the deficit from one being the probability that T does not halt. Concentrate this surplus probability on $P_T(u)$, such that $\sum_{x \in S} P_T(x) = 1$. It can be shown that P is an enumerable probability distribution if and only if $P = \Theta(P_T)$ for some T. In particular, $P_U(x) = \Theta(\mathbf{m}(x))$ for a universal machine U. From this, properties (1), (2), and (3) can be derived.

Levin has shown that Solomonoff's definition, and the two definitions (1) and (2) given above, are equivalent up to a multiplicative constant. Thus, three very different formalizations turn out to define the same notion of universal probability. It is customary in mathematics to view such a circumstance as evidence that we are dealing with a fundamental notion. See [ZL] for the analogous concepts in continuous sample spaces; also see [G2], and [LV1] or [LV2] for elaboration of the cited facts and proofs.

This universal distribution has many important properties. Under $\mathbf{m}$, easily describable objects have high probability, and complex or random objects have low probability. Other things being equal, it embodies Occam's Razor, which says we should prefer simple explanations over complicated ones. To give an example, with $x = 2^n$ we have

$$K(x) \leqq \log n + 2 \operatorname{loglog} n + O(1)$$

and

$$\mathbf{m}(x) = \Omega\left(\frac{1}{n \log^2 n}\right).$$

If we generate the binary representation of y by n tosses of a fair coin, apart from the leading "1," then for the overwhelming majority of outcomes we shall have $K(y) > n$ and $\mathbf{m}(y) = O(2^{-n})$.

By Markov's inequality, for any two probability distributions P and Q, for all k, we have $Q(x) < k \cdot P(x)$ with P-probability at least $1-(1/k)$. By (1) and (3) therefore, for each enumerable probability distribution $P(x)$, we have

$$\sum \{P(x): k\mathbf{m}(x) \geqq P(x) \geqq \mathbf{m}(x)/k\} \geqq 1 - \frac{1}{k}, \tag{4}$$

for all $k \geqq 2^{K(P)+O(1)}$. In this sense, with high P-probability, $P(x)$ is close to $\mathbf{m}(x)$, for each enumerable P. The distribution $\mathbf{m}$ is the only enumerable one (up to order of magnitude) which has that property. In the absence of any a priori knowledge of the actual distribution, therefore, apart from that it is enumerable: studying the behavior under $\mathbf{m}$ is considerably more meaningful than studying the behavior under any other particular enumerable distribution.

DEFINITION. A distribution $P(x)$ is *simple* if it is multiplicatively dominated by some enumerable distribution $Q(x)$, as follows. There is a constant $c \leqq 2^{K(Q)+O(1)}$, such that for all $x \in N$,

$$cQ(x) \geqq P(x). \tag{5}$$

The first question is how large the class of simple distributions is. It certainly includes all enumerable distributions and hence all distributions with bounded precision parameters in our statistics books. We next show that containment is proper and not vacuous.

LEMMA 1. *There is a nonenumerable distribution which is simple.*

Proof. Let A be a subset of the sample space. Consider the distribution

$$P(x) = \begin{cases} c/x^2 & \text{if } x \in A, \\ 0 & \text{otherwise.} \end{cases}$$

The constant c is determined such that $\sum_x P(x) = 1$. Now if we choose A to be not recursively enumerable, then $P(x)$ is not enumerable. But $P(x)$ is multiplicatively dominated by the recursive distribution $Q(x) = 6/(\pi x)^2$. By trivial modification of the above, we can also guarantee that $2Q(x) \geqq P(x)$ for all x. □

LEMMA 2. *There is a distribution which is not simple.*

Proof. We define a probability distribution $f(x)$ which exceeds $\sqrt{x}\,\mathbf{m}(x)$ for infinitely many x. Then by (1) and (5), $f(x)$ is not simple. Let u be the least monotonic upper bound on $\mathbf{m}$, $u(x) = \sup\{\mathbf{m}(y): y \geqq x\}$. Then $u(x) = \Omega(1/\log^2 x)$ (consider the sequence of values $x = 2^n$). The desired function $f(x)$ is defined by $f(x) = u(x)$ for infinitely many x such that $\mathbf{m}(x) \leqq 1/x$, otherwise $f(x) = 0$. We have to set the x's which yield nonzero values for $f(x)$ such that $\sum f(x) \leqq 1$. □

MOTIVATION. If one can dominate the actual distribution by an enumerable distribution, then the theory we develop in this paper can be used to learn. This is the case with all distributions known in statistics, as long as the parameters are computable. What happens when the parameters are real numbers? Let us consider a Bernoulli process $(p, 1-p)$. The probability of k successes out of n outcomes is

$$B(n, k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

If we truncate p to $p - \varepsilon$ in the approximation of a real number p, then the approximating

probability becomes

$$B'(n,k)=\left(1-\frac{\varepsilon}{p}\right)^k\left(1+\frac{\varepsilon}{1-p}\right)^{n-k}B(n,k).$$

Then, for fixed ε, we find that, for $k=0$,

$$\lim_{n\to\infty}\frac{B'(n,0)}{B(n,0)}=\infty.$$

Suppose, however, that we have an algorithm which gives us precision of $A(n,k)$ bits of p, estimating p by $p(n,k)$ such that

$$p-p(n,k)\leqq 2^{-A(n,k)}.$$

Denote the resulting approximation of $B(n,k)$, substituting p by $p(n,k)$, by $\hat{B}(n,k)$. Clearly, for each constant $0<\delta<1$, there is such a function A, which is computable, such that

$$\delta<\frac{\hat{B}(n,k)}{B(n,k)}<\frac{1}{\delta}.$$

In practice, the algorithm giving $A(n,k)$ bits of a real parameter p may consist in estimating p from a large number of outcomes. We cannot guarantee that such a process will always give the required precision. By the law of large numbers, however, it gives the required precision with any required high probability, using sufficiently many outcomes. As it happens, this suffices for the learning application. Similar approximations can be devised for many other distributions with real parameters.

2.1. Simple distribution-independent learning. In this section all concept classes we deal with are over discrete sample spaces. In the learning phase we draw the sample according to **m** instead of according to the underlying probability distribution. However, **m** is positive for all x in N, while the underlying probability distribution often assigns nonzero probability to only a (finite) subset $D\subseteq N$ (like the set of Boolean formulas over n variables). Denote a conditional probability distribution $P(x\mid x\in D)$ by $P(x\mid D)$. If the sample space is N, then we have

$$P(x\mid D)=P(x)\frac{\sum\{P(y): y\in N\}}{\sum\{P(y): y\in D\}}, \tag{6}$$

for $x\in D$, and $P(x\mid D)=0$, otherwise.

We extend the notion of prefix complexity to recursively enumerable sets and enumerable functions. For each set $D\subseteq N$ define:

$$\mathbf{m}(D)=\sum\{\mathbf{m}(y): y\in D\},$$
$$K(D)=-\log \mathbf{m}(D)+O(1).$$

If D is a recursively enumerable set, enumerated by a shortest program p of the reference prefix machine U, then $K(D)=K(p)+O(1)$. Namely, the ith element of D in enumeration order can be described by $0^{l(i)}1iq$, where $U(q)=p$ and therefore

$$\mathbf{m}(D)=\sum\{2^{-2l(i)-1-K(p)+O(1)}: 1\leqq i\leqq |D|\}$$
$$=2^{-K(p)+O(1)}.$$

If f is an enumerable function, enumerated by a program p of the reference prefix machine U, then $K(f)\leqq K(p)+O(1)$.

DEFINITION. Let $P(\cdot \mid D)$ be a simple conditional probability distribution. We say that the learning algorithm *samples according to* $\mathbf{m}$, if in the learning phase of the concept class distributed according to $P(\cdot \mid D)$, the algorithm draws random samples from $\mathbf{m}(\cdot \mid D)$.

All properties of $\mathbf{m}(\cdot)$ versus simple $P(\cdot)$ hold by similar derivation for the conditionals $\mathbf{m}(\cdot \mid D)$ versus simple $P(\cdot \mid D)$. We can formalize the sampling notion in different ways. The following two implementations are equivalent. Assume that D is a recursive set.

(1) The learning algorithm is equipped with an $\mathbf{m}$ oracle that supplies samples according to $\mathbf{m}$. Intuitively, this is natural in a teacher–student situation, or in auto-learning by nonrandom experiments. In such circumstances, samples with low Kolmogorov complexity are drawn with high probability. To obtain $\mathbf{m}(\cdot \mid D)$, we simply draw examples from $\mathbf{m}(\cdot)$ and discard the ones not in D. If we need to draw m examples according to $\mathbf{m}(\cdot \mid D)$, then it suffices to draw $O(2^{K(D)} \cdot m)$ examples under $\mathbf{m}(\cdot)$. Namely, for each $x \in D$,

$$\mathbf{m}(x \mid D) = \mathbf{m}(x) \frac{\sum \{\mathbf{m}(y) \colon y \in N\}}{\sum \{\mathbf{m}(y) \colon y \in D\}}$$

$$= \Theta(2^{K(D)} \cdot \mathbf{m}(x)).$$

(2) The algorithm has access to an $\mathbf{m}$ table in the form of a division of the real half-open interval $[0, 1)$ into nonintersecting half-open subintervals I_x such that $\bigcup I_x = [0, 1)$. For each x, the length of interval I_x is $\mathbf{m}(x)/\sum_y \mathbf{m}(y)$. Define the *cylinder* Γ_r as the set of all infinite binary strings starting with r. To draw a random example from $\mathbf{m}$, the algorithm uses a sequence $r_1 r_2 \cdots$ of outcomes of fair coin flips until the cylinder Γ_r, $r = r_1 r_2 \cdots r_k$, is contained in some interval I_x. It is easy to see that this procedure of selecting x, using a table $\mathbf{m}$ and fair coin flips, is equivalent to drawing an x randomly according to distribution $\mathbf{m}$. To obtain $\mathbf{m}(\cdot \mid D)$, see under item (1).

The table $\mathbf{m}$ is noncomputable. In § 3 we develop time-limited analogues of simple distributions and a corresponding universal distribution. In many learning algorithms we consider only examples of fixed length n, which allows us to precompute a time-limited version of $\mathbf{m}$. Such a table of the time-limited version of $\mathbf{m}$ needs to be precomputed only *once*, and being available, can be used repeatedly by *any* learning process for learning *any* concept class.

Let us discuss the description length of enumerable distributions. By *program length* of each distribution, we mean the length of the Turing machine which computes it plus the length of the description of parameters such as the mean and variance in the normal distribution. Let us look at an example. Let the sample space be N. The uniform distribution L is defined by

$$L(x) = \frac{6 \cdot 2^{-l(x)}}{(\pi \cdot l(x))^2}.$$

Hence the uniform probability on n-length strings is the conditional probability $L(x \mid D) = 2^{-n}$ with $D = \{x \colon l(x) = n\}$. Then,

$$K(L(\cdot \mid D)) \leqq K(L) + 2 \log n + O(1).$$

Our model of learning has the following *completeness property*. Without loss of generality, we assume that the sample space D is a subset of N. The following theorem

says roughly that a concept class is polynomially learnable under $\mathbf{m}$ if and only if it is polynomially learnable under each simple distribution. Its statement is unduly complicated because of some technicalities.

THEOREM 1. *Let C be a concept class, $D \subseteq N$ the associated sample space, $m = \min\{l(s(c)): c \in C\}$, and $d > 0$ a constant. C is polynomially learnable under the universal distribution $\mathbf{m}$ if and only if it is polynomially learnable under each sample distribution $P(\cdot \mid D)$ $(=P(\cdot))$, provided there is an enumerable distribution Q dominating P satisfying $K(Q) \leqq d \log m + O(1)$, and either (i) or (ii) is the case.*

(i) *In the sampling phase, the examples are drawn according to the conditional distribution $\mathbf{m}(\cdot \mid D)$.*

(ii) *$K(D) \leqq d \log m + O(1)$, and in the sampling phase, the polynomially many examples are drawn according to unconditional distribution $\mathbf{m}(\cdot)$ where the degree of the polynomial depends on d. (Typically, there is an n, such that D consists of all n-length vectors over a finite alphabet, so that $K(D) = \log n + 2 \operatorname{loglog} n + O(1)$, and $d = 2$ suffices.)*

Proof. Note, that in the statement of the theorem, the polynomial associated with the learning algorithm depends on $\mathbf{m}$ and the concept class C, but not on the underlying distribution P. The "if" case is vacuous since $\mathbf{m}$ is a simple distribution. We prove the "only if" case.

(i) *Conditional Version.* We prove that if C with sample space D is polynomially learnable under $\mathbf{m}$, then it is learnable under $P(\cdot \mid D)$ while sampling according to $\mathbf{m}(\cdot \mid D)$. Let $c \in C$ be the concept to be learned, and let D be the set of all examples associated with C. Let n denote the length of the representation $s(c)$ of c. Since P is simple, by (5), there is an enumerable Q, and a constant $d_1 < 2^{K(Q)+O(1)}$, such that, for all $x \in N$, we have $d_1 Q(x) > P(x)$. Using (6), and $P(N) = P(D)$, there is a constant $d_2 > 0$,

$$d_2 = d_1 \frac{Q(D)}{Q(N)},$$

$$d_2 Q(x \mid D) \geqq P(x \mid D).$$

Since Q is in turn dominated by $\mathbf{m}$, and by (1) and (3), there is a constant $d_3 = 2^{K(Q)+O(1)}$, possibly dependent on D, such that for all x, we have $d_3 \mathbf{m}(x) \geqq Q(x)$. Using (6) again, there is a constant $d_4 > 0$, such that

$$d_4 = d_3 \frac{\mathbf{m}(D) \cdot Q(N)}{\mathbf{m}(N) \cdot Q(D)},$$

$$d_4 \mathbf{m}(x \mid D) \geqq Q(x \mid D).$$

By (6),

$$d_5 = \frac{\mathbf{m}(N)}{\mathbf{m}(D)}$$

$$\mathbf{m}(x \mid D) = d_5 \mathbf{m}(x).$$

Hence, for all $x \in D$,

$$d_2 d_4 d_5 \mathbf{m}(x) \geqq P(x \mid D)$$

$$d_2 d_4 d_5 \leqq 2^{2K(Q)+K(D)+O(1)}.$$

Assume C is polynomially learnable under $\mathbf{m}$ using learning algorithm A. Let n be the length $l(s(c))$ of the representation $s(c)$ of the concept $c \in C$ to be learned. Then one can run algorithm A with error parameter ε / n^{2d+1} in polynomial time. Let *err* be

the set of strings that are misclassified by the learned concept. So with probability at least $1-\delta$,

$$\sum_{x \in err} \mathbf{m}(x) \leqq \frac{\varepsilon}{n^{2d+1}}.$$

Then, since $m \leqq n$,

$$\sum_{x \in err} P(x \mid D) \leqq 2^{2K(Q)+O(1)} \sum_{x \in err} \mathbf{m}(x \mid D) \leqq \varepsilon,$$

for large enough n.

(ii) *Unconditional Version.* We prove that if C with sample space D, $K(D) \leqq d \log m + O(1)$, is polynomially learnable under $\mathbf{m}$, then it is learnable under $P(\cdot \mid D)$ while sampling according to $\mathbf{m}(\cdot)$. Let Q be as in (i). Since Q is dominated by $\mathbf{m}$, then by (1) and (3), there is a constant $d_6 > 0$, such that

$$d_6 = 2^{K(Q(\cdot \mid D))+O(1)}$$

$$d_6 \mathbf{m}(x) \geqq Q(x \mid D).$$

Hence, for all $x \in N$,

$$d_2 d_6 \mathbf{m}(x) \geqq P(x \mid D).$$

$$d_2 d_6 \leqq d_1 2^{K(Q(\cdot \mid D))+O(1)}.$$

Since $K(Q)$ is a constant independent of n,

$$K(Q(\cdot \mid D)) \leqq K(Q) + K(D) \leqq 2d \log m + O(1).$$

Assume C is polynomially learnable under $\mathbf{m}$, using Learning Algorithm A. Let n be the length $l(s(c))$ of the representation $s(c)$ of the concept $c \in C$ to be learned. Run Algorithm A with error parameter ε / n^{3d+1} in polynomial time, such that, with probability at least $1-\delta$,

$$\sum_{x \in err} P(x \mid D) \leqq n^{3d+1} \sum_{x \in err} \mathbf{m}(x) \leqq \varepsilon,$$

for n large enough. (We oversample polynomially under $\mathbf{m}$ in order to approximate $\mathbf{m}(\cdot \mid D)$.) □

In the next sections we show how to exploit this completeness theorem to obtain new learning algorithms. After all, if we know the sample space has a simple distribution, then we can learn using any learning algorithm for the specific distribution $\mathbf{m}$. The latter distribution has the remarkably convenient property that in a polynomial sample *all* examples of logarithmic complexity occur with probability near one.

Obviously, Theorem 1 also holds if we replace $\mathbf{m}$ by any distribution Q that dominates P. But any such Q which is not equivalent to $\mathbf{m}$ and yet dominates all simple distributions, is not simple itself.

Since $\mathbf{m}$ assigns higher probabilities to simpler strings, one could suspect that after polynomially many examples, *all* simple strings are sampled and the strings that are left unsampled have only very low (inverse polynomial) probability. However, the next theorem shows that this is not the case.

THEOREM 2. *Let S be a set of n^c samples drawn according to* $\mathbf{m}$. *Then*

$$\sum_{x \notin S} \mathbf{m}(x) = \Omega\left(\frac{1}{(\log n)^2}\right). \tag{7}$$

Proof. Consider the first n^{c+2} strings. These strings have Kolmogorov complexity at most $(c+2)\log n + 2 \log\log n + O(1)$ each. The total probability for these strings,

excluding S, is at least

$$(n^{c+2}-n^c)\Omega\left(\frac{1}{n^{c+2}(\log n)^2}\right)=\Omega\left(\frac{1}{(\log n)^2}\right).$$

By using more efficient prefix coding, (7) can be improved to

$$\sum_{x\notin S}\mathbf{m}(x)=\Omega\left(\frac{1}{\log n \log\log n \log\log\log n\cdots}\right). \qquad \square$$

Remark. This says that we cannot just do polynomial sampling and hope to do trivial learning by listing the examples in a table. Namely, the error probability required in Theorem 1 is $\varepsilon/n^{\Omega(1)}$, which cannot be satisfied by (7). So nontrivial learning is required to satisfy the requirements with small ε.

Remark. Since $\mathbf{m}(x)$ is not recursively computable, one may be inclined to suggest that "sampling according to $\mathbf{m}$" has only theoretical interest. In a separate paper, we will investigate the *polynomial-time-bounded approximation* of $\mathbf{m}$ and show that it has similar domination properties with respect to the polynomial-time simple distributions, which still include all textbook distributions we know of. In fact, in all of the discussion below, the Kolmogorov complexity K and the universal distribution $\mathbf{m}$ can be replaced by their polynomial-time-bounded version. For a polynomial-time-bounded version of $\mathbf{m}$, it will be possible to precompute its table once, to be used for sampling in the learning phase.

2.2. Log *n*-DNF. We have established that learning under the universal distribution is important, since if one can polynomially pac-learn under the universal distribution, then one can polynomially pac-learn, using the same algorithms, under any simple distribution by using the universal distribution and sampling according to it. Are there classes of concepts which are not (known to be) learnable under all distributions (in the sense of Valiant) but which are learnable while sampling according to $\mathbf{m}$? We first consider a class for which it is not known whether it is Valiant learnable.

DNF stands for "disjunctive normal form." A DNF is any sum $m_1+m_2+\cdots+m_r$ of monomials, where each monomial m_i is the product of some literals chosen from a universe $x_1,\cdots,x_n$ or their negations $\overline{x_1},\cdots\overline{x_n}$. A k-DNF is a DNF where each monomial consists of at most k literals. It is known that k-DNF is learnable in Valiant's sense [V]. One is inclined to think that also $(n-k)$-DNF is learnable, and so is the sum of monomial terms such that every third variable is true, or every seventh element is true, etc., like $\sum_i x_1x_2\cdots x_{\lfloor n/i\rfloor i}$, where i ranges from 1 to $f(n)$ for some sublinear function f. Or more generally, expressed in a DNF form,

$$\sum_{\phi\in\Phi} x_{\phi(1)}x_{\phi(2)}\cdots x_{\phi(n)}, \tag{8}$$

with Φ a set of total recursive functions such that $|\Phi|$ is polynomial in n, $K(\phi)=O(\log n)$ and $\phi(i)\leqq n$, for $\phi\in\Phi$ and $i=1,\cdots,n$. This class should also be learnable. It is not known whether such formulae are Valiant learnable. We show they are learnable in our sense. For example, the class contains DNF formula like, for any $1\leqq i\leqq n$,

$$f=x_1\cdots x_i+x_{i+1}\cdots x_{2i}+x_{n-i}\cdots x_n.$$

Let us write log n-DNF to denote DNF formulae over n variables, where each monomial term is of Kolmogorov complexity $O(\log n)$, and the length of the formula does not exceed a polynomial in n. This is a superset of k-DNF (it contains all formulae of the form (8)).

THEOREM 3. *The class* log n-DNF *is polynomially learnable under* $\mathbf{m}$.

Proof. Let $f(x_1, \cdots, x_n)$ be a $\log n$-DNF where each term has Kolmogorov complexity at most $c \log n$. If m is a monomial term in f, we write $m \in f$. Sample $n^{c'}$ examples, where we choose c' large enough to satisfy the argument below.

Claim 1. With probability greater than $1 - n^c/e^n$, all examples of the following form will be drawn:

For each monomial term m of f, the example vector that satisfies m and has zero values for all variables not in m, denoted by 0_m; the example vector that satisfies m and has one value for all variables not in m, denoted by 1_m.

Proof. Each monomial m above has Kolmogorov complexity at most $c \log n$, and therefore example 0_m has Kolmogorov complexity at most $c \log n + O(1)$. Therefore, $\mathbf{m}(0_m) \geqq 2^{-c\log n + O(1)} \geqq n^{-c-1}$, for large enough n. This is the probability that 0_m will be sampled. Let E be the event that 0_m does not occur in $n^{c'}$ examples. Then

$$Pr(E) \leqq (1 - n^{-c-1})^{n^{c'}} \leqq \frac{1}{2}\left(\frac{1}{e}\right)^n,$$

for c' large enough. The same estimate holds for the probability that the example 1_m is not sampled in $n^{c'}$ examples. There are only n^c possible monomials m such that $K(m) \leqq c \log n$. Hence, the probability such that all vectors 0_m and 1_m associated with such monomials m are sampled is at least $1 - n^c/e^n$. □

Now we approximate f by the following learning procedure.

LEARNING ALGORITHM.

(0) Sample $n^{c'}$ examples according to $\mathbf{m}$. Let *Pos* (*Neg*) be the set of positive (negative) examples sampled.

(1) For each pair of examples in *Pos*, construct a monomial which contains x_i if both vectors have "1" in position i, contains $\bar{x}_i$ if both vectors have "0" in position i, and does not contain variable x_i otherwise.

(2) Among monomials constructed in step (1) delete the ones that imply examples in *Neg*. The remainder forms a set S.

(3) Let $A_m = \{x: m(x) = 1\}$, that is, A_m is the set of positive examples implied by monomial m. Use a greedy set cover algorithm to find a small set, C, of monomials $m \in S$, such that $\bigcup_{m \in C} A_m$ covers all positive examples in *Pos*.

We have to prove the correctness of the algorithm.

Claim 2. With probability greater than $1 - n^c/e^n$, $\{m: m \in f\} \subseteq S$.

Proof. By Claim 1, with probability at least $1 - n^c/e^n$, all vectors 1_m and 0_m such that monomial m has Kolmogorov complexity at most $c \log n$ are drawn. From 1_m and 0_m, m is formed in step (1) of the algorithm. Thus with probability at least $1 - n^c/e^n$, all monomials of f belong to S. □

Of course, many other monomials may also be in S. Finding all of the original monomials of f precisely is NP-hard. For the purpose of learning, it is sufficient to approximate f. We use the following result due to Johnson [J] and Lovász [Lo].

Claim 3. Given sets $A_1, \cdots, A_n$, such that $\bigcup_{i=1}^n A_i = A = \{1, \cdots, q\}$. If there exist k sets $A_{i_1}, \cdots, A_{i_k}$ such that $A = \bigcup_{j=1}^k A_{i_j}$, then it is possible to find in polynomial time $l = O(k \log q)$ sets $A_{i_1}, \cdots, A_{i_l}$ such that $A = \bigcup_{j=1}^l A_{i_j}$. □

Let f have k monomials. These k monomials cover the positive examples in the sense that $Pos \subseteq \bigcup_{m \in f} A_m$. By Claim 3, we can use about $O(kn)$ monomials to approximate f and cover the examples in *Pos* in polynomial time. Then Occam's Razor Theorem implies that our algorithm polynomially learns $\log n$-DNF. □

Remark. The reader may wonder if the following scheme would work to learn $\log n$-DNF: Code each monomial as binary vectors efficiently. Then since we can

sample all binary vectors of Kolmogorov complexity $c \log n$, decoding these into monomials gives us all monomials of Kolmogorov complexity $c \log n$. Then we run the set cover algorithm to choose a small set of monomials and this will achieve learning. But this scheme will not work since the sampling is done among 2^n Boolean vectors. However, there are 3^n monomials of n variables. It can be easily proved that there is no effective encoding scheme which selectively codes only 2^n monomials including all $c \log n$ Kolmogorov complexity monomials. (Because, if there is such an effective procedure, then this procedure can be used to show that certain strings have Kolmogorov complexity larger than, say, $c \log n$. We know this is not possible [LV1].) An interesting open question remains: is $\log n$-decision list polynomially learnable under $\mathbf{m}(x)$? A $\log n$-decision list is a decision list of Rivest [R] with each term having Kolmogorov complexity $O(\log n)$.

2.3. Simple DNF. We learn a more general class of DNF formulae in this section. This time, we allow each term to have very high Kolmogorov complexity. We call a vector v k-close to a monomial m, if after changing no more than k bits in v, v would satisfy m. Let us define a DNF formula f to be *simple* if, for each term m of f, there is vector v_m that satisfies m but is not 1-close to any other monomials of f and $K(v_m) = O(\log n)$. Simple DNFs can contain many high Kolmogorov complexity terms. An easy example is to take a random binary sequence y with $K(y) \geqq n - c$. Then the number of ones in y is about $n/2$. Construct a term m containing x_i if $y_i = 1$, and neither x_i nor $\bar{x}_i$ otherwise. Then $K(m) \geqq n - c$, but the vector 1_m of all ones satisfying m has $K(1_m) = O(\log n)$. The class of simple DNFs is pretty general. The learning algorithm for the class of simple DNF formulae is as follows, assuming $K(v_m) \leqq c \log n$ for all v_m.

LEARNING ALGORITHM.

(0) Sample n^{c+2} examples from distribution $\mathbf{m}$.

(1) For each positive example e, construct the corresponding monomial, m_e, of size n, which is satisfied only by e.

(2) For each monomial m_e constructed in step (1), mark variable x_i in m_e if there is a negative example that differs with e by only one bit at location i. In m_e delete the unmarked variables. Remove those monomials that are satisfied by some negative examples.

(3) Use the set cover algorithm to choose a small set, S, of monomials that *cover* all the positive examples, i.e., so that each positive example is implied by some monomial in S.

THEOREM 4. *The class of simple* DNF *is polynomially learnable under* $\mathbf{m}$.

Proof. For each monomial m in f, there is a vector v_m that satisfies only m and no other monomials in f. Hence if in v_m we flip a bit corresponding to a variable in m, it becomes a negative example of Kolmogorov complexity $(c+1) \log n$. Therefore it will be sampled with high probability according to Claim 1 in the previous section. From this v_m and corresponding negative examples (which will all be sampled with high probability), one forms precisely m. Note that variable x_i will be marked if and only if it appears in m. There will also be many other monomials so constructed not belonging to f. But since all monomials of f are in the set, we can cover all positive examples using about $l(f)n$ monomials in polynomial time. Hence using Occam's Razor Theorem, we learn correctly with high probability. □

2.4. Monotone k-term DNF. The previous subsections provided several classes that are polynomially learnable under the universal distribution, and hence in our

sense under all simple distributions, and which are not known to be polynomially learnable in the general Valiant model. The purpose of this subsection is to demonstrate a class that was shown to be not polynomially learnable in Valiant's sense, unless RP = NP, but which is polynomially learnable under **m** (and hence under all simple distributions in our model).

A Boolean formula is *monotone* if no literal in it is negated. A k-term DNF is a DNF consisting of at most k monomials. In [PV] it was shown that learning a monotone k-term DNF by k-term (or $2k$-term) DNF is NP-complete (see also [KLPV]).

THEOREM 5. *The class of monotone k-term* DNF *is polynomially learnable by monotone k-term* DNF, *under* **m**.

Proof. Assume we are learning a monotone k-term DNF $f(x_1, \cdots, x_n) = m_1 + \cdots + m_k$, where m_i's are the k monotone monomials (terms) of f.

LEARNING ALGORITHM.

(0) Draw $n^{k'}$ examples according to **m**, for $k' > k+1$. Set DNF $g := \emptyset$. (g is the DNF we will eventually output as approximation of f.)

(1) Pick a positive example $a = (a_1, \cdots, a_n)$. Form a monotone term m such that m includes x_i if $a_i = 1$.

(2) *for* each positive example $a = (a_1, \cdots, a_n)$ *do*: if $a_i = 0$ and deleting x_i from m violates no negative examples, delete x_i from m.

(3) Remove from the sample all positive examples which are implied by m. Set $g \leftarrow g + m$. If there are still positive examples left, then go to step 1, else halt and return g.

We show that the algorithm is correct. Let us write $m_i \subseteq m$ for two monotone monomials if all the variables that appear in m_i also appear in m. At step 1, the monomial m obviously implies no negative examples, since for some monomial m_i of f we must have $m_i \subseteq m$. Step 2 of the algorithm keeps deleting variables from m. If at any time for no monomial $m_i \in f$ holds $m_i \subseteq m$, then there exists a negative example that contains at most k zeros such that it satisfies m but no m_i of f. This negative example is of Kolmogorov complexity at most $k \log n$, hence with high probability (at least $1 - 1/e^n$) is contained in the sample. Hence at step 2, with high probability, there will be an m_i such that $m_i \subseteq m$. Hence we eventually find a correct m_i (precisely) with high probability. Then at step 3, we remove the positive examples implied by this m_i and continue on to find another term of f. The algorithm will eventually output $g = f$ with high probability (at least $1 - (n^c/e^n)$ for some constant c). □

Remark. Note that this is not an approximation algorithm like the ones in the previous sections. This algorithm outputs the precise monotone formula with high probability.

3. Discrete sample spaces and polynomial-time computable distributions. The drawback of the theory developed in § 2 is that **m** is not computable. In this section we scale the entire theory down to a more feasible domain.

Consider the countably infinite discrete sample space N. It is convenient to formulate this section in terms of distribution functions $P^*: \{1, 2, \cdots\} \rightarrow [0, 1]$, where $P^*(x)$ is the probability of all instances not exceeding x. Its density $P(x) = P^*(x) - P^*(x-1)$ is the probability of example x.

A function f is computable in time t, if there exists a Turing machine T which on input x computes output $f(x)$ in at most $t(l(x))$ steps. We construct a time-bounded version of equation (2) as follows. First we define a time-bounded version of Kolmogorov complexity, see also [LV1]. Let U be the reference universal prefix machine

(as in § 2). Let t be a total function. Define the t-time-bounded version of $K(x)$ by:

$$K_t(x)=\min\{l(p)\colon U(p)=x, \text{ the computation taking } \leqq t(l(x)) \text{ steps}\}.$$

Note that, for all at least linear t and x, $K_t(x)\leqq l(x)+2l(l(x))+O(1)$. The limiting value of $K_t(x)$, for $t(\cdot)\to\infty$, is

$$\lim_{t(l(x))\to\infty} K_t(x)=K(x).$$

According to generally accepted notions of feasibility (in the theory of computing), t should be polynomial. Let t denote a *polynomial* in the sequel, up to multiplicative constant factors.

DEFINITION. The t-time-bounded version of $\mathbf{m}$, denoted by $\mathbf{m}_{\langle t\rangle}$, is defined as follows:

$$\mathbf{m}_{\langle t\rangle}(x)=2^{-K_t(x)}$$

$$\mathbf{m}^*_{\langle t\rangle}(x)=\sum_{y\leqq x}\mathbf{m}_{\langle t\rangle}(y).$$

Note that, for all t and x, we have $\mathbf{m}_{\langle t\rangle}(x)\geqq 2^{-l(x)-2l(l(x))+O(1)}$. In the limit, for $t(\cdot)\to\infty$, we have, using (2),

$$\lim_{t(l(x))\to\infty}\mathbf{m}_{\langle t\rangle}(x)=\Theta(\mathbf{m}(x)).$$

Let $\mathbf{P}(t)$ denote the class of t-time-computable probability distributions P^*. We want to show that $\mathbf{m}_{\langle nt(n)\rangle}$ multiplicatively dominates all probability density functions P with $P^*\in\mathbf{P}(t)$.

NOTE. We do *not* know whether $\mathbf{m}^*_{\langle nt(n)\rangle}$ is polynomial-time-computable. We can compute the density function $\mathbf{m}_{\langle n(t(n))\rangle}$ in $nt(n)2^n$-time, by computing $K_{nt(n)}$ by running all programs of length less than $n+O(1)$ for $nt(n)$ steps, and determining the length of the shortest program which halts with output x. A variation yields a similar upper bound on the computation time for $\mathbf{m}^*_{\langle nt(n)\rangle}(x)$, by computing the sum

$$\mathbf{m}^*_{\langle nt(n)\rangle}(x)=\sum_{y\leqq x}\mathbf{m}_{\langle l(y)t(l(y))\rangle}(y).$$

THEOREM 6. *The distribution $\mathbf{m}_{\langle nt(n)\rangle}$ is universal for the class $\mathbf{P}(t)$ in the sense that it multiplicatively dominates each P with $P^*\in\mathbf{P}(t)$: there exists a constant $c>0$, such that, for all x, we have $c\mathbf{m}_{\langle nt(n)\rangle}(x)\geqq P(x)$.*

This follows immediately from the following lemma.

LEMMA 3. *If the probability distribution P^* is computable in time t, then there is a constant c, such that for all x:*

$$K_{nt(n)}(x)\leqq -\log P(x)+c,$$

where $l(x)=n$.

Proof. We wish to show that $K_{nt(n)}(x)\leqq -\log P(x)+c$. Without a polynomial-time bound, a proof similar to that of the optimality of the Shannon–Fano code would be sufficient (like that of (2), see [LV2]). But we have to deal with the time bound here.

We divide the real interval $[0, 1]$ into subintervals such that the code word $p(x)$ for source word x "occupies" $[P^*(x-1), P^*(x)]$. Notice that $\sum_x P(x)\leqq 1$. The *binary interval* determined by the finite binary string r is the half-open interval $[0.r, 0.r+2^{-l(r)})$ corresponding to the set of reals (cylinder) Γ_r consisting of all reals $0.r\cdots$. If Γ_r is the leftmost greatest binary interval contained in $I_x=[P^*(x-1), P^*(x))$, then x is encoded as $p(x):=r$. Since length $l(I_x)=P(x)$, it is easy to show that $l(p(x))\leqq -\log P(x)+2$ bits.

We give polynomial encoding and decoding algorithms. The encoding algorithm is trivial: since P^* is computable in time $t(n)$, given source word x, $l(x)=n$, the code word $p(x)$ can be computed through $P^*(x-1)$ and $P^*(x)$ in $O(t(n))$ time. In order to compute p^{-1}, the decoding function, given a code word $p(x)$, we proceed as follows.

DECODING ALGORITHM.

(1) Set $k:=1$.

(2) Repeatly set $k:=2k$ until $2^{-l(p(x))}$ lays in or left of interval $[P^*(k-1), P^*(k)]$. Set $u:=k$ and $l:=k/2$.

(3) (Binary search) Let $m=(u+l)/2$. If $\Gamma_{p(x)}$ is the leftmost maximum binary interval in $[P^*(m-1), P^*(m)]$, then return $x=m$. Otherwise set $u:=m$ if $2^{-l(p(x))}$ lays left of $P^*(m-1)$ and set $l:=m$ if $2^{-l(p(x))}$ lays right of $P^*(m)$.

This procedure is similar to a binary search, and it takes at most time

$$\sum_{i=0}^{n} O(t(i)) = O(nt(n)), \qquad n=l(x),$$

to find x.

This completes our encoding/decoding of x using distribution P^*. To reconstruct source word x from its code word $p(x)$, with $l(p(x)) \leqq -\log P(x)+2$ by the above construction, the description of the Decoding Algorithm is also needed. If q is a binary program to compute P^*, then the latter description takes $l(q)+O(1)$ bits. Since $K_{nt(n)}(x)$ is the shortest program from which x can be reconstructed in $O(nt(n))$ steps, setting $c=l(q)+O(1)$, we have

$$K_{nt(n)}(x) \leqq -\log P(x) + c. \qquad \square$$

COROLLARY. *Note that* $c=K_{nt(n)}(P)+O(1)$ *suffices, since the program* q *computing* P^* *may be reconstructed in time* $O(n(t(n)))$ *from a shorter description* q'.

Above we noticed that we do not know how to compute $\mathbf{m}_{\langle t\rangle}$ in polynomial time. But for a subset of the domain we can do better.

LEMMA 4. *The probabilities* $\mathbf{m}_{\langle t\rangle}(x)=2^{-K_t(x)}$ *for the set of all* x*'s with* $K_t(x)=O(\log n)$, *can be computed in time polynomial in* $t(n)$.

Proof. As in § 2, use the universal prefix Turing machine U with a one-way input tape, a two-way work tape, and an output tape. On the input tape it is provided with a sequence of 0's and 1's, generated by random flips of a fair coin. The universal machine finds the first initial segment of the coin-tossing sequence which constitutes a program in a prefix-free code. Such a program must be in form of $(n, t(n), p)$. If this is not the case, U discards the code. Otherwise U proceeds as follows. Simulate p for $t(n)$ steps. If p stops within $t(n)$ steps, then print the output of p, otherwise print *undefined* on the output tape. This way the probability of generating a string x, $l(x)=n$, is at least $2^{-K_t(x)-O(\log n)}$.

If $K_t(x)=O(\log n)$, then we can try all programs p of length $l(p)\leqq K_t(x)$, and determine the values of $\mathbf{m}_{\langle t\rangle}(x)=2^{-K_t(x)}$ for all such x together, in time polynomial in $t(n)$. $\square$

In time polynomial in $t(n)$ we can obviously find all x of length n with $K_t(x)=O(\log n)$, and by Lemma 4 also determine their probabilities $\mathbf{m}_{\langle t\rangle}(x)$. This way we can precompute $\mathbf{m}_{\langle t\rangle}$ for the high probability x's in polynomial time. However, for us the following Lemma is more important.

LEMMA 5. *To compute a table* $\mathbf{m}_{\langle t\rangle}(x+1), \cdots \mathbf{m}_{\langle t\rangle}(x+2^n)$, $l(x)=n$, *takes time* $O(t(n)2^n)$. *In other words, we can divide* $[0, 1)$ *into* 2^n *half-open disjoint intervals* I_y *with*

$$|I_y| = \frac{\mathbf{m}_{\langle t\rangle}(y)}{\mathbf{m}^*_{\langle t\rangle}(x+2^n) - \mathbf{m}^*_{\langle t\rangle}(x)},$$

such that $\bigcup_y I_y = [0, 1)$, $y = x+1, \cdots, x+2^n$.

Hence, if we want to learn a concept class using examples of fixed size n, sampling according to $\mathbf{m}_{\langle t\rangle}$, we can *precompute* the interval representation of the table (as in the Lemma) once and for all, and use it to sample according to $\mathbf{m}_{\langle t\rangle}$ by means of a sequence of fair coin flips analogous to what we explained in § 2 for $\mathbf{m}$.

The entire §§2 and 3 can now be reformulated in terms of t-time-limited simple distributions, $\mathbf{m}_{\langle nt(n)\rangle}$, and $K_{nt(n)}$. For example, a distribution $P(x)$ is t-simple if it is dominated by some distribution $Q \in \mathbf{P}(t)$, as follows. There is a constant $c \leqq 2^{K_{nt(n)}(Q)+O(1)}$, such that, for all $x \in N$, we have $cQ(x) > P(x)$. We leave it to the reader to prove the analogons of Lemmas 1 and 2 and Theorems 1-5.

To use it in practice, proceed as follows. Fix a low t, like $t = O(n^2)$, and precompute once and for all an $\mathbf{m}_{\langle n^3\rangle}$ table. Let C be a concept class which is polynomially learnable under $\mathbf{m}_{\langle n^3\rangle}$. For example, C is the class of n^2-simple DNF (analogous to simple DNF in § 2.3). We can use this table, together with random coin flips as explained in § 2, to polynomially learn n^2-simple DNF under all n^2-simple distributions.

4. Continuous sample spaces. We consider continuous spaces, say the set of all one-way infinite sequences over some basic set of elements B. The sample space is $\Omega = B^\infty$. A *measure* μ on Ω satisfies, with Λ the empty word and $x \in B^*$:

$$\mu(\Lambda) = 1, \tag{9}$$

$$\mu(x) = \sum_{a \in B} \mu(xa). \tag{10}$$

The meaning of $\mu(x)$ is the combined probability (measure) of the set of elements, or *cylinder*, $\Gamma_x \subseteq \Omega$ defined as $\Gamma_x = \{x\omega: \omega \in \Omega\}$. Using standard notions from measure theory, we can form the closure of the set of cylinders under complementation and taking countable union (and therefore countable intersections), each resulting set having an appropriate μ-measure. The resulting sets are called Borel sets, and form a so-called σ-algebra denoted by, say, σ. The pair (σ, μ) is called a probability field (see [Ha]).

Example. The discrete probability distributions we considered before, actually probability densities, correspond to measures with $B = N \cup \{u\}$ and consider the measures of the cylinders Γ_a, where $a \in N \cup \{u\}$.

Example. Another example is the Lebesgue measure, or uniform measure, on interval $[0, 1)$. Take $B = \{0, 1\}$, and consider the measure $\lambda(x) = 2^{-l(x)}$. This has a geometric interpretation. Consider real numbers in $[0, 1)$ as being represented by their binary representation. A number like $\frac{1}{2}$ has two representations. Then we choose the representation with infinitely many zeros. The uniform measure of the cylinder Γ_x is the length of the real interval $[0.x, 0.x + 2^{-l(x)})$.

A measure μ over Ω is *enumerable*, if the set

$$\{(x, y): x \in (B - \{u\})^*, y \in Q, y \leqq \mu(x)\} \tag{11}$$

is recursively enumerable, where u is a special "undefined" symbol in B.

It remains to elucidate the role of u. We would have liked to satisfy (9) and (10) with $B = \{0, 1\}$, but (9), (10), and (11) together, with $B - \{u\}$ replaced by B, imply that

the measure is recursive. It can be shown that the set of recursive measures cannot be enumerated such that it contains a universal recursive measure. Using enumerable functions it turns out that we cannot satisfy (9) and (10), but only $\mu(\Lambda) \leqq 1$ and $\mu(x) \geqq \mu(x0)+\mu(x1)$. The surplus probability corresponds with nonterminating computations. By use of the "undefined" symbol u, we normalize the enumerable defective measures to proper measures, by concentrating the surplus probability on dummy "undefined" elements in the sample space.

We obtain an enumeration of enumerable measures as follows. Consider an enumeration $T_1, T_2, \cdots$ of Turing machines, with a one-way input tape, a one-way output tape, and a two-way worktape. The input elements are taken from a set A, one element in each input tape square. Initially, the output tape contains the symbol u in each square. A machine T in the list computes a function from A^* into B^*, as follows. If after having scanned an initial input segment p, upon shifting its input head to the $(l(p)+1)$th input tape square, the initial segment of the output tape up to the position of the output tape head contains x, $l(x) < \infty$, while the output tape does not contain x after the processing of any proper prefix of p, then $T(p) = x$. Such Turing machines are called *monotonic machines*. Setting $A = \{0, 1\}$ and $B = \{0, 1, u\}$, we define

$$\mu_T(x) = \sum_{T(p)=x} 2^{-l(p)}.$$

In other words, if the input to T is supplied by tosses of a fair coin, then $\mu_T(x)$ is the probability that the output of T starts with x. Using the definition of μ_T, it is easy to define a recursive function which approximates μ_T from below in the sense of (11). Hence μ_T is an enumerable measure. It can be shown [ZL] that μ is an enumerable measure if and only if $\mu = \mu_T$ for some monotonic machine T. Hence, we have obtained an enumeration $\mu_1, \mu_2, \cdots$ of enumerable measures.

We are particularly interested in μ_U, where U is a universal monotonic machine U in the list $T_1, T_2, \cdots$. Suppose U has the property that $U(0^{l(l(n))}1l(n)np) = T_n(p)$ for all p in $\{0, 1\}^*$. This means that if $T = T_n$ in the enumeration, then

$$\mu_U(x) \geqq \frac{\mu_T(x)}{2n \log^2 n},$$

for all x in $\{0, 1\}^*$. Fixing U and defining $\mathbf{M} = \mu_U$, we have established the result of Levin: the enumerable measure $\mathbf{M}$ multiplicatively dominates all enumerable measures μ, in the sense that

$$\forall i \in N^+ \exists c > 0 \forall x \in (B - \{u\})^*[c\mathbf{M}(x) \geqq \mu_i(x)]. \tag{12}$$

We call such an $\mathbf{M}$ a *maximal* enumerable measure or a *universal* enumerable measure.

A measure μ over Ω is *simple* if it is dominated multiplicatively by an enumerable measure σ (there is a $c > 0$ such that $\sigma(x) > c\mu(x)$ for all x). Obviously, each enumerable measure is simple, and each simple measure μ is multiplicatively dominated by $\mathbf{M}$ in the sense of $\mu(x) = O(\mathbf{M}(x))$.†

4.1. Learning continuous concepts. A concept class is a subset $C \subseteq 2^\Omega$ of concepts, each of which is a Borel set. If c is a concept to be learned, then $x \in \Omega$ is a positive example if $x \in c$, and it is a negative example if $x \in \Omega - c$. The remaining definitions of learning can now be rephrased in the continuous setting in the obvious way.

†Similar to the discrete case one can show that there are simple measures which are not enumerable, and there are measures which are not simple.

While for discrete sample spaces all concept classes are learnable (although not all are polynomially learnable), this is not the case for continuous sample spaces. Here we show that all continuous concept classes are learnable over each simple measure μ if and only if they are learnable under the universal measure $\mathbf{M}$. In contrast with polynomial learning of discrete concepts, we do not need to require (but do allow) that the learning algorithm samples according to the universal measure.

THEOREM 7. *A concept class C of concepts in Ω is learnable under* $\mathbf{M}$ *if and only if it is learnable under each simple measure.*

Proof. The "if" part holds vacuously. We only need to prove the "only if" part. We use some definitions and results from [BI]. According to [BI], C_ε is an *ε-cover* of C, with respect to distribution μ, if for every $c \in C$ there is a $\hat{c} \in C_\varepsilon$ which is ε-close to c, $(\mu(c \Delta \hat{c}) < \varepsilon)$. A concept class C is *finitely coverable* if for every $\varepsilon > 0$ there is a finite ε-cover C_ε of C, everything with respect to a given measure μ. A finite ε-cover C_ε has finitely many concepts c, and each c is in the closure of the set of cylinders under finite union, complement (and finite intersection).

LEMMA 6. *A concept class C is finitely coverable with respect to μ if and only if C is learnable with respect to μ.*

Proof. We give the main idea of the proof in [BI].

"*Only If.*" Assume that C is finitely coverable under μ. We show that C is learnable under μ. This is done by encoding the finite cover set C' of C into the learning algorithm. We iterate the following procedure. We draw a sample, and choose the concept from C' which minimizes the error in the classification of the elements from the sample. By the standard application of Occam's Razor Theorem this algorithm learns if the size of the sample sufficiently exceeds the size of the concept selected. This can always be guaranteed since there is no priori limit on the sample size, but there is an a priori limit on the size of concepts in C' since C' is finite.

"*If.*" Assume that A learns C under μ using a sample of size l with error less than ε with probability greater than $1-\delta$. We show that C is finitely coverable under μ. Let $n = n(\mu, C, \varepsilon)$ be the cardinality of the smallest 2ε-cover C_n of C under μ (n is possibly infinite).

Choose a set $C_{2\varepsilon} \subseteq C$ of n pairwise 2ε-far concepts ($\mu(c \Delta c') \geqq 2\varepsilon$ for all unequal c, c' in $C_{2\varepsilon}$). For instance, define a sequence of concept classes $C_0, C_1, \cdots$ by $C_0 = \emptyset$ and $C_{i+1} = C_i \cup \{c\}$ such that c is 2ε-far from all concepts in C_i. Then $C_{2\varepsilon} = C_n$ if n is finite, or $C_{2\varepsilon} = \lim_{i \to \infty} C_i$ if $n = \infty$.

Let c be the concept to be learned. Let $\mathbf{x} = (x_1, \cdots, x_l) \in \Omega^l$ be a sequence of l examples and $\mathbf{L} = (L_1, \cdots, L_l) \in \{0,1\}^l$. Then $(\mathbf{x}, \mathbf{L})$ is a sample of size l. If $L_i = 1$ if $x_i \in c$ and $L_i = 0$ if $x_i \in \Omega - c$, then we denote this $\mathbf{L}$ by $\mathbf{L}_c$. Let $h_A(\mathbf{x}, \mathbf{L})$ be the concept returned by learning algorithm A. For $c \in C$, let

$$\chi(c, h_A(\mathbf{x}, \mathbf{L}), \varepsilon) = \begin{cases} 1 & \text{if } \mu(h_A(\mathbf{x}, \mathbf{L}) \Delta c) < \varepsilon, \\ 0 & \text{otherwise.} \end{cases}$$

Consider the sum

$$S = \sum_{c \in C_{2\varepsilon}} \int_{\mathbf{x}} \chi(c, h_A(\mathbf{x}, \mathbf{L}_c), \varepsilon)\, d\mu. \tag{13}$$

By hypothesis, the probability that $\chi(c, h_A(\mathbf{x}, \mathbf{L}_c), \varepsilon) = 1$ exceeds $1-\delta$, for a randomly drawn sample $(\mathbf{x}, \mathbf{L}_c)$ from μ. From (13) we obtain $S > (1-\delta)n$. On the other hand

we have, trivially,

$$S \leqq \sum_{c \in C_{2\varepsilon}} \int_{\mathbf{x}} \sum_{\mathbf{L} \in \{0,1\}^l} \chi(c, h_A(\mathbf{x}, \mathbf{L}), \varepsilon)\, d\mu$$
$$= \int_{\mathbf{x}} \sum_{\mathbf{L} \in \{0,1\}^l} \sum_{c \in C_{2\varepsilon}} \chi(c, h_A(\mathbf{x}, \mathbf{L}), \varepsilon)\, d\mu.$$

Since concepts in $C_{2\varepsilon}$ are 2ε-far, for every $(\mathbf{x}, \mathbf{L})$ there exists at most one $c \in C_{2\varepsilon}$ such that $\chi(c, h_A(\mathbf{x}, \mathbf{L}), \varepsilon) = 1$. Thus

$$(1-\delta)n \leqq S \leqq \int_{\mathbf{x}} \Big(\sum_{\mathbf{L} \in \{0,1\}^l} 1 \Big)\, d\mu = \int_{\mathbf{x}} 2^l\, d\mu = 2^l.$$

Hence $l \geqq \log((1-\delta)n)$. When $n = \infty$, this implies that the Learning Algorithm A has to take an infinite sample. Hence finitely learnable means finitely coverable. □

Using Lemma 6, if C can be learned under $\mathbf{M}$, then C can be finitely covered with respect to $\mathbf{M}$. Let μ be a simple distribution, and $d > 0$ such that $\mathbf{M}(x) \geqq d\mu(x)$ for all x. Then, any finite εd-cover of C, with respect to $\mathbf{M}$, is also a finite ε-cover with respect to μ. Using Lemma 6 again, it follows that C is learnable with respect to μ. □

Remark. Note that this is a strong statement since we are saying that if one can learn under $\mathbf{M}$, then one can also learn under any simple measure μ, while sampling according to μ. In the polynomial learning of discrete concepts we required sampling according to $\mathbf{m}$ in the learning phase. This improvement of Theorem 7 over Theorem 1 is made possible by relaxing the "polynomial learning" requirement to "learning" (without a priori time bound).

Obviously, by the proof, if a concept class C is finitely coverable with respect to $\mathbf{M}$, then it is also finitely coverable and hence learnable under any simple distribution μ.

We may wonder whether Theorem 7 is vacuously true. Namely, do there exist concept classes which are learnable under all simple measures but are not learnable under all measures?

DEFINITION. Given a concept class C and finite $S \subseteq \Omega$, if $\{S \cap c \colon c \in C\} = 2^S$, then we say S is *shattered* by C. The *Vapnik-Chervonenkis* (VC) *dimension* of C is the smallest integer d such that no $S \subseteq \Omega$ of cardinality $d+1$ is shattered by C; if no such d exists then the dimension of C is infinite.

It is a fundamental result of [BEHW], that a class C has finite VC-dimension if and only if it is learnable under arbitrary measures.

THEOREM 8. *There is a concept class that is learnable under all simple measures but not learnable under all measures.*

Proof. Each singleton set $\{\omega\}$, $\omega \in \Omega$, is a Borel set obtainable by countable intersection:

$$\{\omega\} = \bigcap_{n \in N} (\Gamma_{\omega_{1:n}}),$$

and therefore is measurable. There are elements $\omega \in \Omega$ with $\mathbf{M}(\{\omega\}) > 0$. Clearly, there are only countably many such elements. Therefore, modified cylinders in Ω' defined by

$$\Gamma'_x = \Gamma_x - \{\omega \in \Gamma_x : \mathbf{M}(\omega) > 0\},$$

are also Borel sets. Let the concept class C be the class of concepts c, where each c is defined by an index set $I \subseteq B^*$, such that

$$c = \bigcup \{\Gamma'_x \colon x \in I\},$$

satisfying: if $x, y \in I$, $x \neq y$, then either $\mathbf{M}(\Gamma'_x) \geqq 2\mathbf{M}(\Gamma'_y)$ or vice versa. We show that C has infinite VC dimension. For any d consider a set $A \subseteq \Omega'$ of cardinality d. For every subset A' of A we can find a concept $c \in C$ of d cylinders such that $c \cap A = A'$. Therefore, by the result of [BEHW] quoted above, C is not learnable under arbitrary measures.

We now show that C is finitely coverable under $\mathbf{M}$, and hence learnable under $\mathbf{M}$, by Lemma 6. For each $\varepsilon > 0$, there are only finitely many cylinders Γ'_x such that $\mathbf{M}(\Gamma'_x) > \varepsilon/2$. We denote this set of cylinders by A_ε. Given ε, we define an ε-cover of C with respect to $\mathbf{M}$ as follows:

$$C_\varepsilon = \{c' \colon \exists c \in C[c' = \bigcup \{\Gamma'_x \in A_\varepsilon : \Gamma'_x \subseteq c\}]\}.$$

It can be easily verified that C_ε is finite and ε-covers C. Therefore C is finitely coverable with respect to $\mathbf{M}$, and C is learnable with respect to $\mathbf{M}$, by Lemma 6. □

5. Concluding remarks. We have restricted Valiant distribution-independent learning to simple-distribution-independent learning. In the case of discrete sample spaces, we found a "hardest" or "universal" distribution $\mathbf{m}$ which holds all the secrets about polynomial learning of simple concepts in this world. In a sense, we were even more successful for continuous sample spaces. There we found the straight completeness result that a concept is learnable under all simple measures if and only if it is learnable under the particular simple measure $\mathbf{M}$. We demonstrated the use of these completeness results by exhibiting new learning algorithms, new learnable concept classes, and distinctness of our model from Valiant's.

Our approach does have its disadvantages, the most obvious one being that $\mathbf{m}$ is not computable, but only enumerable. It turns out that we can scale down the entire theory as developed to more practically interesting classes of computable distributions, for instance, polynomially computable ones. Treating polynomial-time computable distributions as the analogon of the simple distributions, we encounter the mathematically inelegant drawback that this class does not contain a universal distribution—there is a universal distribution for this class but not in it. It has been shown, however, that the class of polynomial sampleable distributions, as defined in [BCGL], contains a universal distribution. One may also further restrict our assumption to even narrower classes to make the theory practically usable. An entirely similar set of considerations holds for the continuous incarnation $\mathbf{M}$ of $\mathbf{m}$.

It seems likely that many simple concepts previously polynomially unlearnable become polynomially learnable in our model. We have given evidence for this by several examples. Is $\log n$ decision list—in analogy with $\log n$ DNF—polymomially learnable in our model? The connection between our approach of sampling under $\mathbf{m}$ and learning via queries is obvious, but has not been treated here. Many other classes, such as monotone DNF formulae, are also attractive candidates that may be learnable in our model.

We view this paper as another step towards a viable mathematical theory for machine learning in the tradition of Solomonoff, Gold and Valiant, as described in [LV2]. The ultimate aim is a theory supporting machine learning using few examples, rather than polynomially many.

Appendix: Simple reversible languages: We exhibit one more discrete concept class which is polynomially learnable under all simple distributions, and not known to be polynomially learnable under all distributions. A deterministic finite automaton (DFA) $\mathbf{A} = (Q, q_0, F, A, \delta)$ consists of a set of states Q, a finite nonempty input alphabet A,

an initial state q_0 and a set of final states $F \subseteq Q$, and a transition function $\delta: Q \times A \to Q$. Reversible languages were defined by Angulin in [A1] (see also [BF]). **A** is 0-*reversible* if it has only one final state ($|F| = 1$), and its reversal $\mathbf{A}^R$ is deterministic. ($\mathbf{A}^R$ is obtained from **A** by reversing each transition in **A** and exchanging the initial and the final state of **A**.) An alternate definition would be that **A** is 0-*reversible* if it is deterministic with one initial state and one final state and $\delta(q_1, a) = \delta(q_2, a)$ implies $q_1 = q_2$. A language L is 0-reversible if it is accepted by a 0-reversible DFA.

Many languages/DFAs are 0-reversible and of low Kolmogorov complexity. Examples are, for fixed n, the language L_1 = set of strings of length at least n and containing an even number of zeroes, and the language $L_2 = \{0^k 1^j: k, j \geqq n\}$.

A nondeterministic finite automaton (NFA) is like a DFA with q_0 replaced by $I \subseteq Q$, and $\delta: Q \times A \to 2^Q$. We generalize 0-reversibility as follows. A k-reversible DFA is a DFA **A** such that in (the possibly nondeterministic) $\mathbf{A}^R$, if two distinct states q_1, q_2 are initial states or $q_1, q_2 \in \delta(q_3, a)$ for $a \in A$, then no string u of length k satisfies both $\delta(q_1, u) \neq \varnothing$ and $\delta(q_2, u) \neq \varnothing$. This guarantees that any nondeterministic choice in the operation of $\mathbf{A}^R$ can be resolved by looking ahead k symbols past the current one. A language is k-reversible if it is accepted by a k-reversible automaton.

For each fixed n, $L_3 = \{0^k 1^m: k \geqq n, m \geqq 1\}$ and $L_4 = \{0^m 1^k: k \geqq n, m \geqq 1\}$ are 1-reversible and have $O(\log n)$ Kolmogorov complexity. We say a *path* from the initial state to a final state is *simple* if it has Kolmogorov complexity $O(\log n)$. A k-reversible DFA **A** is *simple* if each state of **A** lies on a simple path.

Example. We show that the set of k-reversible DFAs of Kolmogorov complexity $O(\log n)$ are simple. To see this, consider **A** such that $K(\mathbf{A}) = c \log l(\mathbf{A})$ where $l(\mathbf{A})$ is the description length of **A**. We show that every state of **A** is on a simple path of Kolmogorov complexity at most $(c+1) \log l(\mathbf{A})$. Without loss of generality, assume that every state of **A** is reachable from the initial state. If this is not the case, we can just delete those obsolete states from **A**. We have assumed that **A** can be specified in $c \log n$ bits. Fix an enumeration of the paths from the initial state to final states of **A** such that each path contains at least one more new state. There are at most $n = l(\mathbf{A})$ such paths since each path must contain at least one new state which is not contained in the previous paths. Obviously, each such path can be specified using **A**, that is, $c \log n$ bits, and the index of the path in $\log n$ bits. Hence the Kolmogorov complexity of each such path is at most $c \log n + \log n$. Every state is on at least one of these paths by construction.

Note that if $K(\mathbf{A}) = O(\log n)$, it is still possible that **A** may have very random paths. For example, the automaton which accepts all strings of length n has Kolmogorov complexity $O(\log n)$, but it actually contains a path for every string of length n. In particular, it contains a path of Kolmogorov complexity n. On the other hand, one can construct (left to the reader) a simple 0-reversible DFA which has Kolmogorov complexity much larger than $\log n$ (like $\Omega(\log^2 n)$).

In the general Valiant distribution-independent setting, it is not known whether the class of 0-reversible languages is learnable. Angluin [A1] shows that the set of k-reversible languages can be identified in the *limit* in the Gold paradigm. A DFA is polynomially learnable by membership queries and equivalence queries [A].

THEOREM 9. *The class of simple k-reversible automata is polynomially learnable under* **m**.

Proof. We first show how to learn a simple 0-reversible DFA under the universal distribution.

Claim 1. The class of 0-reversible DFA of Kolmogorov complexity $O(\log n)$ is polynomially learnable under **m**.

Proof. The following algorithm and proof use ideas in [A1].

LEARNING ALGORITHM.

(1) Randomly sample n^{c+2} positive examples. Construct the trivial tree DFA from these examples.
(2) Merge all the final states in above tree.
(3) *repeat*
    if there are states $p, q \in Q$ such that on input $a \in A$, p, q lead to the same state, then merge p and q
  until no more merges.

We prove that this algorithm correctly infers the underlying DFA **A** with high probability. Each positive example represents a path from q_0 to q_f, where some states may be repeated because of loops in **A**.

By previous arguments, we know that with high probability (at least $1-1/e^{\Omega(n)}$) each string corresponding to a simple path is sampled.

Claim 1.1. Given all simple paths of **A**, **A** can be inferred from the above algorithm.

Proof. All states of **A** are presented at least once in the tree constructed above. It is up to the algorithm to merge them correctly. Now between any two simple paths, P_1 and P_2, if there is a transition from a state a on P_1 to a state b on P_2, then the path from q_0 to a via P_1 then to b then to q_f via P_2 is also simple, hence also given, hence the above merging process will add a transition from a to b correctly. Since this applies to all transitions, eventually **A** will be correctly inferred. As for the nonsimple strings, some of which may also be sampled since they have higher than $1/(\log n)^2$ probability in total, will also fit into the structure. Notice that all above merges do not introduce mistakes since we are dealing with 0-reversible DFAs. □

This also finishes the proof of Claim 1. □

Claim 2. For each k, the class of simple k-reversible DFA is polynomially learnable under **m**.

Proof. A generalization of the algorithm and the proof in Claim 1 shows that simple k-reversible languages are polynomially learnable under **m**, for each fixed k. We refer the readers to [A1] for more details. □

We now show how to learn the class of simple k-reversible languages for all k. The algorithm is given as follows:

for $k := 1$ *to* ∞ *do*

(1) Apply the algorithm for learning k-reversible language to learn a k-reversible DFA;
(2) Draw a polynomial, in the size of above derived DFA, number of examples, according to $\mathbf{m}(x)$, to test the inferred automaton;
(3) *if* the DFA learned is consistent with $(1-\varepsilon/2)$ fraction of the test set *then* output this DFA
  else continue with the next k value;

At step (3), if the DFA is consistent with $(1-\varepsilon/2)$ fraction of the test set, then applying Occam's Razor Theorem, the DFA we have learned approximates the real DFA with high probability. Note that it is not necessary that the real k-reversible DFA is inferred. The rest of the correctness and complexity analysis are standard and similar to that in [A1], so we again refer the reader to [A1]. □

Acknowledgments. Peter Gács, Gloria Kissin, Ray Solomonoff, and John Tromp commented on the manuscript. John suggested the need for Lemma 2; Peter Gács suggested turning semimeasures like **m** and **M** into measures by concentrating the

surplus probability on an undefined symbol. We thank the referees for their valuable comments.

REFERENCES

[A] D. ANGLUIN, *Learning regular sets from queries and counter-examples*, Tech. Report TR-464, Computer Science Department, Yale University, New Haven, CT, 1986.

[A1] ———, *Inference of reversible languages*, J. Assoc. Comput. Mach., 29 (1982), pp. 741-765.

[A2] ———, *On the complexity of minimum inference of regular sets*, Inform. and Control, 39 (1978), pp. 337-350.

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

[BI] G. BENEDEK AND A. ITAI, *Learnability by fixed distributions*, in Proc. First ACM Workshop on Computational Learning Theory, 1988, pp. 80-90.

[BF] A. BIERMANN AND J. FELDMAN, *On the synthesis of finite-state machines from samples of their behavior*, IEEE Trans. Comput., C-21 (1972), pp. 592-597.

[BEHW] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. WARMUTH, *Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension*, in Proc. 18th ACM Symposium on Theory of Computing, 1986, pp. 273-282.

[BEHW1] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. WARMUTH, *Occam's Razor*, Inform. Process. Lett., 24 (1987), pp. 377-380.

[G] E. GOLD, *Complexity of automaton identification from given data*, Inform. and Control, 37 (1978), pp. 302-320.

[G1] P. GÁCS, *On the symmetry of algorithmic information*, Soviet Math. Dokl., 15 (1974), pp. 1477-1481; a correction to this article appeared in Soviet Math. Dokl., 15 (1974), p. 1481.

[G2] ———, *Lecture notes on descriptional complexity and randomness*, manuscript, Department of Computer Science, Boston University, Boston, MA, October 1987.

[HA] P. R. HALMOS, *Measure Theory*, Springer-Verlag, Berlin, 1974.

[HLW] D. HAUSSLER, N. LITTLESTONE, AND M. WARMUTH, *Expected mistake bounds for on-line learning algorithms*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 100-109.

[J] D. JOHNSON, *Approximation algorithms for combinatorial problems*, J. Comput. System Sci., 9 (1974), pp. 256-276.

[KL] M. KEARNS AND M. LI, *Learning in the presence of malicious errors*, in Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 267-280.

[KLPV] M. KEARNS, M. LI, L. PITT, AND L. G. VALIANT, *On the learnability of Boolean formulae*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 285-295.

[KV] M. KEARNS AND L. VALIANT, *Cryptographic limitations on learning Boolean formulae and finite automata*, in Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 433-444.

[L] L. A. LEVIN, *Laws of information conservation (nongrowth) and aspects of the foundation of probability theory*, Problems Inform. Transmission, 10 (1974), pp. 206-210.

[LV1] M. LI AND P. VITÁNYI, *Kolmogorov complexity and its applications*, in Handbook on Theoretical Computer Science, Vol. 1, Jan van Leeuwen, ed., North-Holland, Amsterdam, 1990, pp. 187-254.

[LV2] ———, *Inductive reasoning and Kolmogorov complexity*, in Proc. Fourth IEEE Structure in Complexity Theory Conference, 1989, pp. 165-185.

[L] N. LITTLESTONE, *Learning quickly when irrelevant attributes abound: A new linear threshold algorithm*, in Proc. 28th IEEE Symposium on the Foundations of Computer Science, 1987, pp. 68-77.

[Lo] L. LOVÁSZ, *On the ratio of optimal integral and fractional covers*, Discrete Math., 13 (1975), pp. 383-390.

[N] B. K. NATARAJAN, *On learning Boolean functions*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 296-304.

[PV] L. PITT AND L. G. VALIANT, *Computational limitations on learning from examples*, J. Assoc. Comput. Mach. 35 (1989), pp. 965-984.

[PW] L. PITT AND M. WARMUTH, *Reductions among prediction problems*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 60-69.

[PW1] ———, *The minimum consistent* DFA *problem cannot be approximated within any polynomial*, in Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 421-432.

[R] R. RIVEST, *Learning decision-lists*, Machine Learning, 2 (1987), pp. 229–246.

[RS] R. RIVEST AND R. SCHAPIRE, *Diversity-based inference of finite automata*, in Proc. 28th IEEE Symposium on the Foundations of Computer Science, 1987, pp. 78–88.

[SO] R. SOLOMONOFF, A *formal theory of inductive inference, Parts* 1, 2, Inform. and Control, 7 (1964), pp. 1–22, 224–254.

[V] L. G. VALIANT, *A Theory of the Learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[ZL] A. K. ZVONKIN AND L. A. LEVIN, *The complexity of finite objects and development of the concepts of information and randomness by means of the theory of algorithms*, Russian Math. Surveys, 25:6 (1970), pp. 83–124.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 936-950, October 1991
© 1991 Society for Industrial and Applied Mathematics
008

ON THE COMMUNICATION COMPLEXITY OF SOLVING A POLYNOMIAL EQUATION*

ZHI-QUAN LUO†‡ AND JOHN N. TSITSIKLIS†

Abstract. This paper considers the problem of evaluating a function $f(x, y)$ ($x \in \Re^m$, $y \in \Re^n$) using two processors P_1 and P_2, assuming that processor P_1 (respectively, P_2) has access to input x (respectively, y) and the functional form of f. A new general lower bound is established on the communication complexity (i.e., the minimum number of real-valued messages that have to be exchanged). The result is then applied to the case where $f(x, y)$ is defined as a root z of a polynomial equation $\sum_{i=0}^{n-1} (x_i + y_i) z^i = 0$ and a lower bound of n is obtained. This is in contrast to the $\Omega(1)$ lower bound obtained by applying earlier results of Abelson.

Key words. communication complexity, polynomial equation, lower bound

AMS(MOS) subject classifications. 05, 68

1. Introduction. In a computer network where a set of processors wishes to perform some computational task, communication can sometimes become a bottleneck, especially when communication resources are scarce. This is particularly so in the area of parallel and VLSI computation (see, e.g., [BT89], [U84]), where the communication issues have been studied extensively. In such contexts, it is desirable to design algorithms that require as little information exchange as possible. Problems of minimizing the amount of exchanged information also arise in the context of decentralized signal processing, where each local processor collects some partial data to be processed collectively. In this paper, we study the "communication complexity" (i.e., the minimum possible amount of information exchange) of some particular computational tasks.

Generally speaking, communication complexity depends both on the topology of a computer network and on the nature of the computational task under consideration. In this paper, we ignore the topological issues by assuming that there are only two processors, say P_1 and P_2. We use the following model of communications introduced by Abelson [A80]. Let there be given a continuously differentiable function $f: D_x \times D_y \mapsto \Re$, where D_x and D_y are some open subsets of $\Re^m$ and $\Re^n$, respectively. It is assumed that processor P_1 (respectively, P_2) has access to a vector $x \in D_x$ (respectively, $y \in D_y$) and the formula defining f. The processors P_1, P_2 proceed to evaluate $f(x, y)$ by exchanging messages, using a *two-way communication protocol*, in which messages can be sent in both directions. Let us use π to denote a two-way communication protocol and $r(\pi)$ to denote the number of messages exchanged in π. In addition, let $T_{1\to 2}$ (respectively, $T_{2\to 1}$) denote the set of indices i for which the ith message is sent from P_1 to P_2 (respectively, from P_2 to P_1). The protocol π consists of $r(\pi)$ functions $m_1, \cdots, m_{r(\pi)}: D_x \times D_y \mapsto \Re$, with $m_i(x, y)$ being interpreted as the value of the ith message. These message functions must depend on the inputs x and y in a very special way. Precisely, for each i, there must exist some real-valued function $\hat{m}_i$ such that

$$(1.1) \quad m_i(x, y) = \hat{m}_i(x, m_1(x, y), \cdots, m_{i-1}(x, y)) \quad \forall (x, y) \in D_x \times D_y \quad \text{if } i \in T_{1\to 2},$$

* Received by the editors July 17, 1989; accepted for publication (in revised form) December 5, 1990. This research was supported by Office of Naval Research contract N00014-84-K-0519 (NR649-003) and Army Research Office contract DAAL03-86-K-0171.

† Operations Research Center and Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

‡ Present address, Department of Electrical and Computer Engineering, Room 225/CRL, McMaster University, Hamilton, Ontario, L8S 4L7, Canada.

or

$$(1.2)\quad m_i(x, y) = \hat{m}_i(y, m_1(x, y), \cdots, m_{i-1}(x, y)) \quad \forall (x, y) \in D_x \times D_y \quad \text{if } i \in T_{2\to 1}.$$

Furthermore, we require that either:

(a) There exists a function h such that

$$(1.3)\qquad f(x, y) = h(x, m_1(x, y), \cdots, m_{r(\pi)}(x, y)) \quad \forall (x, y) \in D_x \times D_y,$$

(this corresponds to the case where processor P_1 performs the final computation) or

(b) There exists a function h such that

$$(1.4)\qquad f(x, y) = h(y, m_1(x, y), \cdots, m_{r(\pi)}(x, y)) \quad \forall (x, y) \in D_x \times D_y,$$

which corresponds to the case where processor P_2 computes the final result.

Typically, some smoothness constraints are imposed on the functions m_i, $\hat{m}_i$, and h. For example, [A80] considers the class of two-way communication protocols (denoted by $\Pi_2(f; D_x \times D_y)$) in which the functions m_i, $\hat{m}_i$, and h are twice continuously differentiable. In this paper, we consider a more general class of protocols in which the message functions m_i, $\hat{m}_i$ are once continuously differentiable and the final evaluation function h is continuous. We denote this class of two-way protocols for computing f by $\Pi_1(f; D_x \times D_y)$. We define the two-way communication complexity of computing f with protocols in $\Pi_2(f; D_x \times D_y)$ as

$$C_2(f; D_x \times D_y) = \inf_{\pi \in \Pi_2(f;\, D_x \times D_y)} r(\pi).$$

We define the quantity $C_1(f; D_x \times D_y)$ similarly. Notice that $\Pi_2(f; D_x \times D_y) \subset \Pi_1(f; D_x \times D_y)$. Thus, $C_2(f; D_x \times D_y) \geqq C_1(f; D_x \times D_y)$. As discussed in [L89], $\Pi_1(f; D_x \times D_y)$ is, in some sense, the most general class of protocols for which the notion of communication complexity is well defined for problems involving continuous variables.

A general lower on $C_2(f; D_x \times D_y)$ was established in the fundamental work of Abelson [A80]. In particular, let $f: D_x \times D_y \mapsto \Re$ be a twice continuously differentiable function and let $H_{xy}(f)$ denote the matrix (of size $m \times n$) whose (i, j)th entry is given by $\partial^2 f/\partial x_i \partial y_j$. The following result was proved in [A80].

THEOREM 1.1. *For any $p \in D_x \times D_y$, we have*

$$C_2(f; D_x \times D_y) \geqq \operatorname{rank}(H_{xy}(f))(p).$$

Note that Theorem 1.1 only takes into account the second-order derivatives of f and ignores the derivatives of other orders. Thus, this bound should not be expected to be tight, as was shown in [LT89].

In this paper, we derive a new general lower bound. Our result (Theorem 2.1) makes use of the first-order derivatives of f and is fairly intuitive, but surprisingly difficult to prove. Our work was motivated from the problem of distributed computation of a root of a polynomial equation of degree $n-1$. We apply our result to this problem and obtain a lower bound of n, in contrast to the $\Omega(1)$ lower bound obtained from Abelson's result. In [L89], a similar $\Omega(n)$ lower bound is established for the same problem, but under a more restricted class of communication protocols in which the functions m_i, $\hat{m}_i$ $(i = 1, \cdots, r(\pi))$ are assumed to be polynomials. The proof in [L89] makes use of a result from dimension theory and is algebraic in nature, in contrast to the analytic approach in the proof given here.

In related work ([LT89]), Abelson's result has been extended by considering a more restricted class of communication protocols; in particular, some improved lower

bounds on one-way and two-way communication complexity have been obtained by exploiting the algebraic structure present in certain problems. Communication complexity has also been studied under discrete communication models (see, e.g., [MS82], [PS82], [PT82], [Y79]). In these models, the messages are no longer real numbers, but binary strings. A substantial amount of research has been devoted to the study of the communication complexity of selected combinatorial problems ([AU83], [PE86], [U84]). A different model is introduced in [TL87] for the problem of approximately minimizing the sum of two convex functions under the assumption that each convex function is known to a different processor.

The rest of this paper is organized as follows. In § 2, we prove our main result (Theorem 2.1). In § 3, we apply the result of § 2 to establish a lower bound of n for the problem of computing a root of a polynomial equation of degree $n-1$. In § 4, we compare our result with Abelson's. Finally, the Appendix contains certain results from multidimensional calculus that are needed in § 2.

2. Main result. Let $f: D_x \times D_y \mapsto \Re$ be a continuously differentiable function, where D_x and D_y are some open subsets of $\Re^m$ and $\Re^n$, respectively. We use the notation $\nabla_x f(x, y)$ (respectively, $\nabla_y f(x, y)$) to denote the m-dimensional (respectively, n-dimensional) vector whose components are the partial derivatives of f with respect to the components of x (respectively, y). Also, for any set $S \subset D_x$, we use $[\nabla_y f(x, y); x \in S]$ to denote the subspace of $\Re^n$ spanned by the vectors $\nabla_y f(x, y)$, $x \in S$. Finally, for any set $S \subset D_y$, $[\nabla_x f(x, y); y \in S]$ is similarly defined.

ASSUMPTION 2.1. *For any $y \in D_y$, we let*

$$\mathcal{S}^{(2)}(y) = \{S \subset D_x \mid f(S, y) \text{ contains an open interval}\}.^1$$

(*For any $x \in D_x$, $\mathcal{S}^{(1)}(x)$ is similarly defined.*)

(a) *For any $y \in D_y$ and any nonempty open set $S \subset D_x$, we have $S \in \mathcal{S}^{(2)}$.*

(b) *For any $x \in D_x$ and any nonempty open set $S \subset D_y$, we have $S \in \mathcal{S}^{(1)}$.*

(c) *For some nonnegative integer n_f, we have*

$$\dim [\nabla_y f(x, y); x \in S] \geqq n_f \quad \forall y \in D_y \quad \forall S \in \mathcal{S}^{(2)}(y). \tag{2.1}$$

(d) *For some nonnegative integer m_f, we have*

$$\dim [\nabla_x f(x, y); y \in S] \geqq m_f \quad \forall x \in D_x \quad \forall S \in \mathcal{S}^{(1)}(x). \tag{2.2}$$

Our main result is the following.

THEOREM 2.1. *Under Assumption* 2.1, *the following is true*

$$C_1(f; D_x \times D_y) \geqq \min \{n_f, m_f\}. \tag{2.3}$$

The proof of Theorem 2.1 is a long and tedious argument based primarily on elementary differential geometry. Before proving Theorem 2.1, we first give a sketch of the basic proof ideas.

Consider an optimal protocol described by (1.1)-(1.2). By symmetry, we can assume that the final evaluation of $f(x, y)$ is performed by processor P_1, in which case the last message must have been transmitted by processor P_2.

We assume, in order to derive a contradiction, that the number r of messages in the protocol satisfies $r < n_f$. Let us fix a "crossing message sequence" $c = (c_1, \cdots, c_r)$,

[1] The notation $f(S, y)$ stands for the set $\{f(x, y) \mid x \in S\}$. Similar notation will be used later without further comment.

that is, the values $c_i = m_i(x, y)$ of the messages under some execution of the protocol. Fixing c imposes the following constraints on x and y:

$$c_i = \hat{m}_i(x, c_1, \cdots, c_{i-1}), \qquad i \in T_{1\to 2}, \tag{2.4}$$

$$c_i = \hat{m}_i(y, c_1, \cdots, c_{i-1}), \qquad i \in T_{2\to 1}. \tag{2.5}$$

Note that these constraints decouple and can be expressed in the form $x \in S_x(c)$ and $y \in S_y(c)$. With some technical work (making sure that certain Jacobians are nonsingular), we can show that $S_x(c), S_y(c)$ are "smooth" (continuously differentiable) surfaces, depending smoothly on c.

The equation

$$f(x, y) = h(x, m_1(x, y), \cdots, m_r(x, y)) \tag{2.6}$$

shows that $f(x, y)$ depends on y through at most r functions. Taking derivatives and using the chain rule, we can show that for any y^*, and any crossing sequence c, the collection of vectors $\{\nabla_y f(x, y^*) \mid x \in S_x(c)\}$ spans a subspace of dimension at most r.

Note that if $y^* \in S_y(c)$, then $f(x, y^*) = h(x, c)$ for all $x \in S_x(c)$. We consider two cases.

Case 1. If there exists some open set of c's in which $h(x, c) = h(c)$ (i.e., independent of x), for all $x \in S_x(c)$, then there exists an open ball in which the equation $f(x, y) = h(m_1(x, y), \cdots, m_r(x, y))$ holds. But this would imply that $f(x, y)$ could have been evaluated by processor P_2 before transmitting the message $m_r(x, y)$, and we would have a protocol with $r-1$ messages, a contradiction.

Case 2. If Case 1 does not hold, a technical argument shows that there exists some particular c for which $h(x, c)$ is not independent of x. By continuity, $\{h(x, c) \mid x \in S_x(c)\}$ contains an open interval. Hence, $S_x(c)$ belongs to $\mathcal{S}^{(2)}(y^*)$. Therefore, using Assumption 2.1 (d) and the fact that the subspace spanned by the vectors $\{\nabla_y f(x, y^*) \mid x \in S_x(c)\}$ has dimension at most r, we have $n_f \leqq r$, which contradicts our earlier assumption.

To turn the above intuitive argument into a rigorous proof, we have to make sure that all the functions involved are properly defined and have the desired differentiability properties. The rest of this section is devoted to a formal proof of Theorem 2.1.

Let $r = C_1(f; D_x \times D_y)$. We first prove that it is sufficient to show the lower bound (2.3) under the additional assumption

$$r = \min_{\bar{D}_x, \bar{D}_y} C_1(f; \bar{D}_x \times \bar{D}_y), \tag{2.7}$$

where the minimum is taken over all nonempty open subsets $\bar{D}_x, \bar{D}_y$ of D_x, D_y, respectively. Suppose that we have already shown that Theorem 2.1 is true under the assumption (2.7). Let us now show that (2.3) is valid when (2.7) does not hold. In this case, there exists some $r' < r$ and some open subsets $\hat{D}_x \times \hat{D}_y$ of $D_x \times D_y$ such that

$$r' = C_1(f; \hat{D}_x \times \hat{D}_y) = \min_{\bar{D}_x, \bar{D}_y} C_1(f; \bar{D}_x \times \bar{D}_y),$$

where the minimum is taken over all nonempty open subsets $\bar{D}_x, \bar{D}_y$ of $\hat{D}_x, \hat{D}_y$. Thus (2.7) holds with r, D_x, and D_y replaced by $r', \hat{D}_x$, and $\hat{D}_y$, respectively. Since any nonempty open subset of $\hat{D}_x$ (respectively, $\hat{D}_y$) is also a nonempty subset of D_x (respectively, D_y), we see that Assumption 2.1 remains valid (with the same constants n_f, m_f) when D_x, D_y are replaced by $\hat{D}_x, \hat{D}_y$. Therefore, Theorem 2.1 applies and shows that $r > r' \geqq \min\{n_f, m_f\}$, which shows that Theorem 2.1 holds regardless of assumption (2.7).

In the rest of the proof, we will assume that (2.7) holds. Let us consider a protocol that uses exactly r messages, described by (cf. § 1)

$$m_i(x, y) = \hat{m}_i(x, m_1(x, y), \cdots, m_{i-1}(x, y)) \quad \forall (x, y) \in D_x \times D_y \quad \text{if } i \in T_{1\to 2}, \tag{2.8}$$

$$m_i(x, y) = \hat{m}_i(y, m_1(x, y), \cdots, m_{i-1}(x, y)) \quad \forall (x, y) \in D_x \times D_y \quad \text{if } i \in T_{2\to 1}, \tag{2.9}$$

where each m_i and $\hat{m}_i$ is a continuously differentiable function. By symmetry, we can assume that the final evaluation of f is performed by processor P_1. Thus there exists some continuous function h such that

$$f(x, y) = h(x, m_1(x, y), \cdots, m_r(x, y)) \quad \forall (x, y) \in D_x \times D_y. \tag{2.10}$$

Before presenting the main line of argument, we derive three lemmas. Let $u = (x, y)$ and let $D = D_x \times D_y$. Write $m(u) = (m_1(u), \cdots, m_r(u))$ and let $\nabla m(u)$ be the $(m+n) \times r$ matrix whose ith column is the gradient vector $\nabla m_i(u)$, $i = 1, \cdots, r$. Define

$$k = \max_{u \in D} \operatorname{rank} [\nabla m(u)]. \tag{2.11}$$

LEMMA 2.1. $k = r$.

Proof. We show this by contradiction. Suppose that $r > k$. Consider the continuously differentiable mapping $m: D \mapsto \Re^r$, where $D = D_x \times D_y$ is an open set and $m(u) = (m_1(u), \cdots, m_r(u))$. We claim that $\nabla m_1(x, y)$ is not identically zero on the set D. Indeed, if this was the case, then $m_1(x, y)$ would be equal to a constant on the set D, and the first message in the protocol would be redundant. Thus, there would exist a protocol that uses $r-1$ messages, contradicting definition of r. We can therefore apply Theorem A.2 in the Appendix (with the correspondence $m \leftrightarrow F$, $D \leftrightarrow Q$, $r \leftrightarrow s$) to conclude that there exists some positive integer i and some continuously differentiable function g such that

$$m_{i+1}(u) = g(m_1(u), \cdots, m_i(u)) \quad \forall u \in \bar{D}, \tag{2.12}$$

where $\bar{D}$ is some nonempty open subset of D. By taking a subset of $\bar{D}$ if necessary, we can assume that $\bar{D}$ is of the product form $\bar{D}_x \times \bar{D}_y$, where $\bar{D}_x$ and $\bar{D}_y$ are some open subsets of D_x and D_y, respectively. Then, (2.12) would imply that the $(i+1)$st message $m_{i+1}(x, y)$ is redundant for computing f over $\bar{D}_x \times \bar{D}_y$, which contradicts the definition of r (cf. (2.7)). □

Loosely speaking, Lemma 2.1 tells us that each message in an optimal protocol has to contain some "new information" and therefore the corresponding gradient vectors have to be linearly independent. Before we go on to the next lemma, we introduce some more notations. Let $\bar{D}_x \subset D_x$, $\bar{D}_y \subset D_y$ be nonempty open sets such that $\nabla m(u)$ has full rank for every $u \in \bar{D}_x \times \bar{D}_y$. (Such sets can be taken nonempty due to Lemma 2.1, and open due to the continuity of $\nabla m(u)$.) We use $\bar{D}$ as a short notation for $\bar{D}_x \times \bar{D}_y$. Furthermore, for any vector $c = (c_1, \cdots, c_r) \in \Re^r$ and for $i \leqq r$, we let $c^i = (c_1, c_2, \cdots, c_i)$. Let also r_1 (respectively, r_2) be the number of messages sent by processor P_1 (respectively, P_2). In addition, we use the notation $[\nabla_x m_i(x, y); i \in T_{1\to 2}]$ to denote the $m \times r_1$ matrix whose column vectors are $\nabla_x m_i(x, y) = (\partial m_i(x, y)/\partial x_1, \cdots, \partial m_i(x, y)/\partial x_m)$, $i \in T_{1\to 2}$. The $n \times r_2$ matrix $[\nabla_y m_i(x, y); i \in T_{2\to 1}]$ is defined similarly. As a refinement of Lemma 2.1, we have the following lemma.

LEMMA 2.2. *For any* $(x, y) \in \bar{D}$, *we have*

$$\operatorname{rank} [\nabla_x \hat{m}_i(x, c^{i-1}); i \in T_{1\to 2}] = r_1,$$

and

$$\operatorname{rank} [\nabla_y \hat{m}_i(y, c^{i-1}); i \in T_{2\to 1}] = r_2,$$

where $c = m(x, y)$.

Proof. By Lemma 2.1, we see that the matrix $\nabla m(x, y)$ has full rank (and its rank is equal to r) over the set $\bar{D}$. Note that by possibly reindexing the columns of the matrix $\nabla m(x, y)$, we can write $\nabla m(x, y)$ in the form

$$\nabla m(x, y) = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where $A_{11} = [\nabla_x m_i(x, y);\ i \in T_{1\to 2}]$ and $A_{22} = [\nabla_y m_i(x, y);\ i \in T_{2\to 1}]$. From (2.8)-(2.9), it is easily seen that for each $i \in T_{2\to 1}$, there exists a continuously differentiable function M_i such that

$$m_i(x, y) = M_i(y, \{m_l(x, y): l < i, l \in T_{1\to 2}\}), \qquad i \in T_{2\to 1}. \tag{2.13}$$

(In other words, a message sent by processor P_2 can be expressed as a function of y and the messages already received.) By differentiating (2.13), we obtain

$$\nabla_x m_i(x, y) = \sum_{l \in T_{1\to 2}, l < i} d_l(x, y) \nabla_x m_l(x, y), \qquad i \in T_{2\to 1}, \tag{2.14}$$

where each $d_l(x, y)$ is a suitable scalar. Thus,

$$\nabla_x m_i(x, y) \in \operatorname{span}\{\nabla_x m_l(x, y);\ l \in T_{1\to 2}\} \quad \forall (x, y) \in \bar{D} \quad \forall i \in T_{2\to 1}.$$

This means that the columns of A_{12} belong to the span of the columns of A_{11} and therefore

$$\operatorname{rank}[A_{11} \quad A_{12}] = \operatorname{rank}(A_{11}) \leqq r_1.$$

Similarly, one can show that

$$\operatorname{rank}[A_{21} \quad A_{22}] = \operatorname{rank}(A_{22}) \leqq r_2.$$

On the other hand,

$$\begin{aligned} r &= r_1 + r_2 \\ &\geqq \operatorname{rank}(A_{11}) + \operatorname{rank}(A_{22}) \\ &= \operatorname{rank}[A_{11} \quad A_{12}] + \operatorname{rank}[A_{21} \quad A_{22}] \\ &\geqq \operatorname{rank}\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\ &= \operatorname{rank}[\nabla m(x, y)] \\ &= r \quad \forall (x, y) \in \bar{D}. \end{aligned}$$

This implies that

$$\operatorname{rank}(A_{11}) = \operatorname{rank}[\nabla_x m_i(x, y);\ i \in T_{1\to 2}] = r_1$$

and

$$\operatorname{rank}(A_{22}) = \operatorname{rank}[\nabla_y m_i(x, y);\ i \in T_{2\to 1}] = r_2.$$

To show that $\operatorname{rank}[\nabla_x \hat{m}_i(x, c^{i-1});\ i \in T_{1\to 2}] = r_1$, we differentiate (2.8) to obtain

$$\nabla_x m_i(x, y) = \nabla_x \hat{m}_i(x, c^{i-1}) + \sum_{l=1}^{i-1} \frac{\partial \hat{m}_i}{\partial m_l}(x, c^{i-1}) \nabla_x m_l(x, y) \quad \text{if } i \in T_{1\to 2}, \tag{2.15}$$

where $c = m(x, y)$ and $(x, y) \in \bar{D}$. Using (2.14), we see that

$$\sum_{l=1}^{i-1} (\partial \hat{m}_i / \partial m_l)(x, c^{i-1}) \nabla_x m_l(x, y)$$

can be written as a linear combination of the vectors $\{\nabla_x m_l(x, y);\ l < i-1, l \in T_{1\to 2}\}$. Therefore, (2.15) shows that

$$[\nabla_x \hat{m}_i(x, c^{i-1});\ i \in T_{1\to 2}] = [\nabla_x m_i(x, y);\ i \in T_{1\to 2}]\, C = A_{11} C,$$

where C is some upper triangular matrix whose diagonal entries are equal to 1. Hence $\text{rank}\,[\nabla_x \hat{m}_i(x, c^{i-1});\, i \in T_{1\to 2}] = \text{rank}\,(A_{11}) = r_1$. The equality

$$\text{rank}\,[\nabla_y \hat{m}_i(y, c^{i-1});\, i \in T_{2\to 1}] = r_2$$

can be shown by a similar argument. □

Let us fix some more notations. For any vector $c = (c_1, \cdots, c_r) \in \Re^r$ and $1 \leqq i \leqq n$, we let

$$\begin{aligned} &S(c) = \{(x, y) \in D_x \times D_y \mid m_i(x, y) = c_i,\ i = 1, \cdots, r\}, \\ &S_x(c) = \{x \in D_x \mid \hat{m}_i(x, c^{i-1}) = c_i,\ \forall i \in T_{1\to 2}\}, \\ &S_y(c) = \{y \in D_y \mid \hat{m}_i(y, c^{i-1}) = c_i,\ \forall i \in T_{2\to 1}\}, \\ &R^r = \{(m_1(x, y), \cdots, m_r(x, y)) \mid (x, y) \in D_x \times D_y\}. \end{aligned} \tag{2.16}$$

LEMMA 2.3. *For any* $c \in R^r$, *we have*

$$S(c) = S_x(c) \times S_y(c). \tag{2.17}$$

Proof. We have, using definition (2.16) and (2.8)-(2.9),

$$\begin{aligned} S(c) &= \{(x, y) \in D_x \times D_y \mid \hat{m}_i(x, c^{i-1}) = c_i, \forall i \in T_{1\to 2},\ \hat{m}_i(y, c^{i-1}) = c_i, \forall i \in T_{2\to 1}\} \\ &= S_x(c) \times S_y(c). \end{aligned}$$ □

We now fix some $(x^*, y^*) \in \bar{D}$ and let $c^* = m(x^*, y^*)$. Let us define

$$F_i(x, c) = \hat{m}_i(x, c^{i-1}) - c_i \quad \forall c \in R^r, \qquad x \in D_x, \qquad i \in T_{1\to 2}.$$

Thus $F_i(x^*, c^*) = 0$ for all $i \in T_{1\to 2}$. Moreover, it follows from Lemma 2.2 that the matrix $[\nabla_x F(x^*, c^*)]$ has full rank. It is now clear that we are in a position to apply Theorem A.3 in the Appendix (with the correspondence that $u \leftrightarrow x$, and $v \leftrightarrow c$) to conclude that there exist an open subset U_1 of $\Re^r$ containing c^*, and an open subset $\hat{D}_x$ of D_x containing x^*, such that $S_x(c) \cap \hat{D}_x$ is nonempty and connected for all $c \in U_1$. Following a symmetrical argument, we see that there exist open subsets $U_2 \subset \Re^r$ and $\hat{D}_y \subset D_y$ such that $c^* \in U_2$, $y^* \in \hat{D}_y$, and $S_y(c) \cap \hat{D}_y$ is nonempty and connected for all $c \in U_2$. Let $U = U_1 \cap U_2$. Clearly, U is nonempty, since $c^* \in U$. In light of Lemma 2.3, we see that for all $c \in U$,

$$\begin{aligned} \hat{S}(c) &\triangleq S(c) \cap (\hat{D}_x \times \hat{D}_y) \\ &= (S_x(c) \cap \hat{D}_x) \times (S_y(c) \cap \hat{D}_y), \end{aligned}$$

and the set $\hat{S}(c)$ is nonempty and connected. Let us use $\hat{S}_x(c)$ and $\hat{S}_y(c)$ to denote the sets $S_x(c) \cap \hat{D}_x$ and $S_y(c) \cap \hat{D}_y$, respectively.

We now proceed to prove Theorem 2.1. Since we have assumed that the final result is evaluated by processor P_1, it follows that the last message $m_r(x, y)$ must have been sent by processor P_2. (Otherwise, processor P_1 would be able to evaluate $f(x, y)$ on the basis of $m_1(x, y), \cdots, m_{r-1}(x, y)$, and we would have a protocol with $r-1$ messages, thus contradicting (2.7).) Suppose that there exists some function $w: U \mapsto \Re$ such that

$$h(x, c) = w(c) \quad \forall c \in U \quad \forall x \in \hat{S}_x(c), \tag{2.18}$$

where h is the function given by (2.10). We claim that w is a continuous function of c in U. In fact, let c be an arbitrary vector in U and let $\{c_i \in U;\, i = 1, 2, \cdots\}$ be a sequence of vectors converging to c. By Theorem A.3 in the Appendix, we can pick a

convergent sequence of vectors $\{x_i \in \hat{S}_x(c_i); i=1,2,\cdots\}$ such that $\lim_{i\to\infty} x_i = x$ for some $x \in \hat{D}_x$. By using (2.18) and the continuity of h, we see that

$$\lim_{i\to\infty} w(c_i) = \lim_{i\to\infty} h(x_i, c_i) = h(x, c),$$

which implies that w is continuous on U. Since for any $(x, y) \in m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$ we have $m(x, y) \in U$, (2.18) yields

$$f(x, y) = h(x, m(x, y)) = w(m(x, y)) \quad \forall (x, y) \in \hat{D}_x \times \hat{D}_y.$$

Thus f can be evaluated on the basis of $m(x, y)$ alone over the set $m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$ and this can be done by processor P_2 before sending the last message. Thus (2.18) leads to a protocol with $r-1$ messages for computing f over $m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$. This will contradict (2.7) once we show that $m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$ is a nonempty open set. To this effect, we notice that $\hat{S}(c)$ is nonempty and that

$$\hat{S}(c) \subset m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y) \quad \forall c \in U,$$

from which it follows that $m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$ is nonempty. Furthermore, $m^{-1}(U)$ is open since it is the inverse image of the open set U under a continuous mapping. Thus, $m^{-1}(U) \cap (\hat{D}_x \times \hat{D}_y)$ is open, since $\hat{D}_x \times \hat{D}_y$ is open by construction.

Since no function w can have the property (2.18), we conclude that there exists some $\hat{c} \in U$ such that $h(x, \hat{c})$ is a nonconstant function of x on the set $\hat{S}_x(\hat{c})$. Since h is a continuous function and the set $\hat{S}_x(\hat{c})$ is nonempty and connected, we see that $h(\hat{S}_x(\hat{c}), \hat{c})$ must contain an open interval in $\Re$. Using the fact $f(x, y) = h(x, \hat{c})$ for all $(x, y) \in \hat{S}_x(\hat{c}) \times \hat{S}_y(\hat{c})$, we have

$$f(\hat{S}_x(\hat{c}), y) = h(\hat{S}_x(\hat{c}), \hat{c}) \quad \forall y \in \hat{S}_y(\hat{c}).$$

Therefore, $f(\hat{S}_x(\hat{c}), y)$ contains an open interval, or equivalently, $\hat{S}_x(\hat{c}) \in \mathcal{S}^{(2)}(y)$ for all $y \in \hat{S}_y(\hat{c})$ (cf. definition 2.1). Let us fix some $\hat{y} \in \hat{S}_y(\hat{c})$. Then, using the definition of n_f (2.1), there exist $x^1, \cdots, x^{n_f} \in \hat{S}_x(\hat{c})$ such that $\nabla_y f(x^1, \hat{y}), \cdots, \nabla_y f(x^{n_f}, \hat{y})$ are linearly independent. Meanwhile, we observe that

$$\hat{S}_y(\hat{c}) = \{y \in \hat{D}_y \mid \hat{m}_i(y, \hat{c}^{i-1}) = \hat{c}_i \quad \forall i \in T_{2\to 1}\}$$

and that, for any fixed $x \in \hat{S}_x(\hat{c})$, $f(x, y) = h(x, \hat{c})$ is a constant function of y on the set $\hat{S}_y(\hat{c})$. Moreover, by Lemma 2.2, we have

$$\text{rank}\,[\nabla_y \hat{m}_i(y, \hat{c}^{i-1}); i \in T_{2\to 1}] = r_2 \quad \forall y \in \hat{D}_y. \tag{2.19}$$

Thus we are now in a position to apply Theorem A.4 (with the correspondence $A \leftrightarrow \hat{S}_y(\hat{c})$, $F \leftrightarrow \{\hat{m}_i(y, \hat{c}^{i-1}) - \hat{c}_i; i \in T_{2\to 1}\}$) and conclude that

$$\nabla_y f(x, \hat{y}) \in \text{span}\,\{\nabla_y \hat{m}_i(\hat{y}, \hat{c}^{i-1}), i \in T_{2\to 1}\} \quad \forall x \in \hat{S}_x(\hat{c}).$$

Since each $x^j \in \hat{S}_x(\hat{c})$, we see that $\nabla_y f(x^j, \hat{y})$ is the span of the vectors $\{\nabla_y \hat{m}_i(\hat{y}, \hat{c}^{i-1}), i \in T_{2\to 1}\}$, for $j = 1, \cdots, n_f$. Using the fact that the vectors $\nabla_y f(x^j, \hat{y})$ are linearly independent, we conclude that $r \geqq r_2 \geqq n_f \geqq \min\{m_f, n_f\}$, which is the desired result, under the assumption that processor P_1 performs the final evaluation of f. A similar argument yields $r \geqq r_1 \geqq n_f \geqq \min\{m_f, n_f\}$ for the case where processor P_2 performs the final evaluation of f. This completes the proof of the theorem.

As a remark, we note that in the preceding proof we have actually shown that $r_2 \geqq n_f$ in the case where processor P_1 performs the final computation and $r_1 \geqq m_f$ if processor P_2 performs the final computation. Therefore, if $C_1(f; D_x \times D_y) = \min\{m_f, n_f\}$, then either $r_1 = m_f$ and $r_2 = 0$, or, $r_1 = 0$ and $r_2 = n_f$. This means that our

lower bound is tight only for those problems for which one-way communication protocols are optimal.

COROLLARY 2.1. *If $C_1(f; D_x \times D_y) = \min\{n_f, m_f\}$, then any optimal communication protocol for computing f over $D_x \times D_y$ is necessarily a one-way communication protocol.*

3. Computing a root of a polynomial. We now apply Theorem 2.1 to the distributed computation of a root of a polynomial. We shall demonstrate that in this case Abelson's result is far from optimal.

Let $x = (x_0, \cdots, x_{n-1}) \in \Re^n$ and $y = (y_0, \cdots, y_{n-1}) \in \Re^n$; let $F(z; x, y)$ be the polynomial in the scalar variable z defined by

$$F(z; x, y) = \sum_{i=0}^{n-1} (x_i + y_i) z^i. \tag{3.1}$$

Processor P_1 (respectively, P_2) has access to the vector x (respectively, y); and the objective is the computation of a particular root of the polynomial $F(z; x, y)$. In order for the problem to be well defined, we must specify which one of the $n-1$ roots of the polynomial is to be computed. This is accomplished as follows. We fix some $(x^*, y^*) \in \Re^{2n}$ such that one of the roots (call it z^*) of the polynomial $F(z; x^*, y^*)$ is real and simple. This root will vary continuously and will remain a real and simple root as x and y vary in some open set containing x^*, y^*. We formulate this discussion in the following result.

LEMMA 3.1. *Suppose that z^* is a real and simple root of $F(z; x^*, y^*)$. Then, there exist open sets $D_x, D_y \subset \Re^n$ such that $(x^*, y^*) \in D_x \times D_y$ and an infinitely differentiable function $f: D_x \times D_y \mapsto \Re$ such that $f(x^*, y^*) = z^*$ and*

$$F(f(x, y); x, y) = 0 \quad \forall (x, y) \in D_x \times D_y. \tag{3.2}$$

Proof. Note that $(\partial F/\partial z)(z^*; x^*, y^*) \neq 0$, since z^* is a simple root. By the implicit function theorem ([S65, p. 41]), we see that there exists an open set D containing (x^*, y^*) and an infinitely differentiable function $g: D \mapsto \Re$ such that $g(x^*, y^*) = z^*$ and $F(g(x, y); x, y) = 0$ for all $(x, y) \in D$. Now by the continuity of $(\partial F/\partial z)(z; x, y)|_{z=g(x,y)}$ at the point (x^*, y^*), there exist open sets D_x, D_y such that $(x^*, y^*) \in D_x \times D_y \subset D$ and such that $(\partial F/\partial z)(z; x, y)|_{z=g(x,y)} \neq 0$ for all $(x, y) \in D_x \times D_y$. As a result, $g(x, y)$ is a simple root of the polynomial equation $F(z; x, y) = 0$ for all $(x, y) \in D_x \times D_y$. Let f be the restriction of g on $D_x \times D_y$. Clearly, f has all the desired properties. □

By Lemma 3.1, we see that $f(x, y)$ is a root of $F(z; x, y)$ and is a well-defined smooth map from $D_x \times D_y$ to $\Re$. We are interested in the communication complexity $C_1(f; D_x \times D_y)$ of computing $f(x, y)$ as (x, y) varies in the set $D_x \times D_y$. We start by pointing out that Abelson's lower bound (Theorem 1.1) is rather weak.

LEMMA 3.2. *The rank of the matrix $H_{xy}(f)$, whose (i, j)th entry is equal to $\partial^2 f/\partial x_i \partial y_j$, is at most 3, for any $(x, y) \in D_x \times D_y$.*

Proof. We have

$$\sum_{i=0}^{n-1} (x_i + y_i)(f(x, y))^i = 0 \quad \forall (x, y) \in D_x \times D_y.$$

We differentiate both sides of the above equation, with respect to y_m, to obtain

$$\sum_{i=1}^{n-1} i(x_i + y_i)(f(x, y))^{i-1} \cdot \frac{\partial f(x, y)}{\partial y_m} + (f(x, y))^m = 0 \quad \forall (x, y) \in D_x \times D_y, \qquad 0 \leqq m \leqq n-1. \tag{3.3}$$

We differentiate (3.3) further, with respect to x_l, to obtain

$$\begin{aligned}\sum_{i=1}^{n-1} i(i-1)(x_i+y_i)(f(x,y))^{i-2}\frac{\partial f(x,y)}{\partial x_l}\frac{\partial f(x,y)}{\partial y_m}\\ +\sum_{i=1}^{n-1} i(x_i+y_i)(f(x,y))^{i-1}\frac{\partial^2 f(x,y)}{\partial x_l \partial y_m}\\ +m(f(x,y))^{m-1}\frac{\partial f(x,y)}{\partial x_l}+l(f(x,y))^{l-1}\frac{\partial f(x,y)}{\partial y_m}=0,\\ \forall (x,y)\in D_x\times D_y, \qquad 0\leqq m, l\leqq n-1.\end{aligned} \tag{3.4}$$

Since $f(x, y)$ is a simple root, it follows that $\sum_{i=1}^{n-1} i(x_i+y_i)(f(x,y))^{i-1}\neq 0$. Equation (3.4) shows that $\partial^2 f(x,y)/\partial x_l \partial y_m$ is of the form $u_1(l)v_1(m)+u_2(l)v_2(m)+u_3(l)v_3(m)$, where $u_i(l)$, $v_i(m)$ are some real numbers depending on x, y. Therefore the rank of the matrix $H_{xy}(f)$ can be at most 3, for any point $(x,y)\in D_x\times D_y$. □

We now illustrate the power of our general results by deriving a lower bound that matches the obvious upper bound.

THEOREM 3.1. *Let D_x, D_y be as in Lemma* 3.1. *Then*, $C_1(f(x,y); D_x\times D_y)=n$.

Proof. The upper bound $C_1(f; D_x\times D_y)\leqq n$ is obvious, so we concentrate on the proof of the lower bound. To this effect, we will employ Theorem 2.1 and it suffices to verify that Assumption 2.1 holds with $n_f=m_f=n$. Since the roots of a polynomial equation cannot remain constant when the coefficients vary over an open set, it follows that the continuous function $f(x, y)$ given by Lemma 3.1 satisfies parts (a) and (b) of Assumption 2.1. Now we fix some $y\in D_y$ and some $S\in\mathcal{S}^{(2)}(y)$, that is, $S\subset D_x$ and $f(S, y)$ contains an open interval. Let $c_1,\cdots,c_n$ be some distinct real numbers in $f(S, y)$ and $x^1,\cdots,x^n\in S$ such that

$$f(x^i, y)=c_i, \qquad i=1,\cdots,n. \tag{3.5}$$

Let x_j^i be the jth coordinate of x^i. Using (3.3), we see that

$$a_i\nabla_y f(x^i, y)=-\begin{bmatrix}1\\ c_i\\ \vdots\\ c_i^{n-1}\end{bmatrix}, \tag{3.6}$$

where $a_i=\sum_{j=1}^{n-1} j(x_j^i+y_j)c_i^{j-1}$. If we form a matrix whose columns are the vectors $(1, c_i,\cdots,c_i^{n-1})$, $i=1,\cdots,n$, this matrix is a Vandermonde matrix and is nonsingular, because the values $c_1,\cdots,c_n$ are chosen to be distinct. Then, (3.6) implies that the vectors $\nabla_y f(x^i, y)$, $i=1,\cdots,n$, are linearly independent. This proves that $n_f=n$. The proof that $m_f=n$ is similar. □

As a remark, we point out that Theorem 3.1 is in some sense the strongest result possible. The only assumptions we used in showing Theorem 3.1 are that (a) the message functions are continuously differentiable; (b) the final evaluation function is a continuous function; (c) the protocol computes a root of a polynomial on some open set. As discussed in [L89], assumption (a) is necessary since its removal could lead to unreasonable conclusions. Assumption (b) is basic and natural since the function to be computed, i.e., a particular real simple root of some polynomial, is continuous, while assumption (c) is minimal. Finally, we note that no truly two-way communication protocol can be optimal. In other words, if each processor transmits at least one message, then at least $n+1$ messages have to be exchanged. This is a simple consequence of Corollary 2.1 of § 2.

4. Comparison with Abelson's bound. In the previous section, we saw that Theorem 2.1 can yield a much better bound than Abelson's result (Theorem 1.1). However, it is not true, as we shall see next, that Theorem 2.1 always provides a stronger lower bound. The reason is, loosely speaking, that our result only places a constraint on the minimum number of messages that has to be sent by a single processor, while Abelson's result is a bound on the total number of messages sent by both processors. As pointed out at the end of § 2, any two-way communication protocol that attains the lower bound in Theorem 2.1 is necessarily a one-way protocol. Notice that our result makes use of information about the first-order derivatives of function f. This is in contrast to Abelson's result which uses only the second-order derivatives of f. In what follows, we provide an example where Abelson's bound is more effective than our bound.

Let $f(x, y) = x^TQy$, where Q is some $m \times n$ matrix, $x \in \Re^m$ and $y \in \Re^n$. By Theorem 1.1, we see that $C_2(f; \Re^m \times \Re^n) \geqq \text{rank}(Q)$. Using the singular value decomposition of Q, one can construct a protocol that uses exactly rank(Q) messages (see [LT89]). Therefore, we conclude that $C_2(f; \Re^m \times \Re^n) = \text{rank}(Q)$. To see what lower bounds are provided by Theorem 2.1, we need to calculate the values of m_f and n_f.

Suppose that rank$(Q) = r > 0$. Let D_x, D_y be some convex open subsets of $\Re^m$ and $\Re^n$, respectively. We assume that $0 \notin D_x$ and $0 \notin D_y$, in which case $f(x, y)$ is nonconstant as x or y vary in an open subset of D_x or D_y, respectively. Thus parts (a) and (b) of Assumption 2.1 are satisfied. We now show that Assumption 2.1 can only hold with $\min\{m_f, n_f\} \leqq 2$. By the singular value decomposition, there exist two linearly independent families of vectors $u_1, \cdots, u_r$ in $\Re^m$ and $v_1, \cdots, v_r$ in $\Re^n$, such that

$$Q = u_1v_1^T + u_2v_2^T + \cdots + u_rv_r^T. \tag{4.1}$$

It follows that $x^TQy = \sum_{i=1}^r (u_i^Tx)(v_i^Ty)$. Since $r > 0$, there exists some point $(x_0, y_0) \in D_x \times D_y$ such that $x_0^TQy_0 \neq 0$. Hence, we can, without loss of generality, assume that $(u_r^Tx_0)(v_r^Ty_0) \neq 0$. Let $S = \{x \in D_x \mid u_i^Tx = u_i^Tx_0, 1 \leqq i \leqq r-1\}$. Clearly, S is nonempty since $x_0 \in S$. We claim that if $r > 1$, then $f(S, y_0)$ contains an open interval. In fact, (4.1) shows that

$$\begin{aligned} x^TQy_0 &= \sum_{i=1}^{r} (u_i^Tx)(v_i^Ty_0) \\ &= \sum_{i=1}^{r-1} (u_i^Tx_0)(v_i^Ty_0) + (u_r^Tx)(v_r^Ty_0) \quad \forall x \in S. \end{aligned} \tag{4.2}$$

Since u_r is linearly independent from $u_1, \cdots, u_{r-1}$, we see that u_r^Tx is a nonconstant function of x on S. Using (4.2) and the fact that $v_r^Ty_0 \neq 0$, we see that x^TQy_0 is also a nonconstant function of x on the set S. Note that S is connected because D_x is assumed to be convex. It follows that $f(S, y_0)$ contains an open interval. To see that $n_f \leqq 2$, we note that

$$\nabla_y f(x, y_0) = \sum_{i=1}^{r-1} (u_i^Tx_0)v_i + (u_r^Tx)v_r \quad \forall x \in S.$$

Hence, $\dim[\nabla_y f(x, y_0); x \in S] \leqq 2$. Thus Assumption 2.1 can only hold with $n_f \leqq 2$. The relation $m_f \leqq 2$ can be established in a symmetrical fashion. As a result, we have shown that $\min\{m_f, n_f\} \leqq 2$.

Thus, for the problem $f(x, y) = x^TQy$, Theorem 2.1 provides a lower bound of at most 2, as opposed to the lower bound of rank(Q) provided by Abelson's result. Hence, Theorem 2.1 can be quite far from optimal, in general. Furthermore, the above example and the results of § 3 illustrate that Theorems 1.1 and 2.1 are incomparable.

Appendix. This appendix contains some results concerning multivariable functions that are used in § 2.

Let $F: U \times V \mapsto \Re^s$ be a continuously differentiable mapping, where U and V are open subsets of $\Re^r$ and $\Re^t$, respectively. We assume that $r > s$ and that $\text{rank}\,[\nabla_u F(u^*, v^*)] = s$, for some $(u^*, v^*) \in U \times V$. Then, the matrix $\nabla F_u(u^*, v^*)$ has s linearly independent rows and we can find a set $J \subset \{1, \cdots, r\}$ of indices, of cardinality s, such that the vectors $(\partial F_1(u^*, v^*)/\partial u_i, \cdots, \partial F_s(u^*, v^*)/\partial u_i)$, $i \in J$ are linearly independent. We define the projection $\Pi: \Re^r \mapsto \Re^{r-s}$ by letting $\Pi(u)$ be the vector with coordinates u_i, $i \notin J$. We have the following lemma.

LEMMA A.1. *There exists a connected open subset R of $U \times V$, and a connected open set $S \subset \Re^{r+t}$, and a continuously differentiable function $g: S \mapsto R$ such that $(u^*, v^*) \in R$,*

$$S = \{(F(u, v), \Pi(u), v) \mid (u, v) \in R\},$$

and such that

$$(u, v) = g(F(u, v), \Pi(u), v) \quad \forall (u, v) \in R. \tag{A.1}$$

Proof. Consider the mapping $q: U \times V \mapsto \Re^{r+t}$ defined by $q(u, v) = (F(u, v), \Pi(u), v)$. We claim that $\nabla q(u^*, v^*)$ has full rank. To see this, let us permute the rows of $\nabla q(u^*, v^*)$ so that the last $r + t - s$ rows correspond to the partial derivatives with respect to the variables v and u_i, $i \notin J$. Then, $\nabla q(u^*, v^*)$ will have the structure

$$\nabla q(u^*, v^*) = \begin{bmatrix} A & 0 \\ B & I \end{bmatrix},$$

where A, B are suitable submatrices of $\nabla F(u^*, v^*)$ and I is the $(r+t-s) \times (r+t-s)$ identity matrix. Each one of the s rows of matrix A is a vector of the form $(\partial F_1(u^*, v^*)/\partial u_i, \cdots, \partial F_s(u^*, v^*)/\partial u_i)$, $i \in J$, and these vectors are linearly independent by construction. Thus $\det(\nabla q(u^*, v^*)) = \det(A) \neq 0$. The result then follows from the inverse function theorem [S65, p. 35]. □

THEOREM A.1. *Let Q be an open subset of $\Re^r$. Let $F: Q \mapsto \Re^s$ be a continuously differentiable mapping such that*

$$\max_{z \in Q} \text{rank}\,(\nabla F(z)) = s. \tag{A.2}$$

Suppose that $f: Q \mapsto \Re$ is a continuously differentiable function with the property

$$\nabla f(z) \in \text{span}\,\{\nabla F(z)\} \quad \forall z \in Q.$$

Then, there exists some continuously differentiable function h such that $f(z) = h(F(z))$ for all $z \in R$, where R is some open subset of Q.

Proof. Suppose that $z^* \in Q$ is a vector at which the maximum in (A.2) is attained. By taking $t = 0$ and dropping the set V, we see that all the assumptions of Lemma A.1 are satisfied[2], and thus Lemma A.1 applies. Let R, S, and g be as in Lemma A.1. By assumption, $\nabla f(z) \in \text{span}\,\{\nabla F(z)\}$, for all $z \in R$. Thus, for every $z \in R$, there exists a vector $d(z) \in \Re^s$ such that

$$\nabla f(z) = \nabla F(z) d(z) \quad \forall z \in R. \tag{A.3}$$

[2] We have assumed that $r > s$ here. The proof for the case $r = s$ is essentially the same except that Π is redundant.

Using Lemma A.1, we have

$$F(z) = F(g(F(z), \Pi(z))) \quad \forall z \in R,$$

or

$$u = F(g(u, v)) \quad \forall (u, v) \in S. \tag{A.4}$$

Let $\nabla_v g$ be the $(r-s) \times r$ matrix of the partial derivatives of g, with respect to the components of v. Since the left hand side of (A.4) does not depend on v, the chain rule yields

$$0 = \nabla_v g(u, v) \cdot \nabla F(g(u, v)) \quad \forall (u, v) \in S. \tag{A.5}$$

We use Lemma A.1 once more to obtain

$$f(z) = f(g(F(z), \Pi(z))) \quad \forall z \in R.$$

We define a function $\bar{h} : S \mapsto \Re$ by letting

$$\bar{h}(u, v) = f(g(u, v)) \quad \forall (u, v) \in S. \tag{A.6}$$

Note that $\bar{h}$ is continuously differentiable. Using the chain rule,

$$\nabla_v \bar{h}(u, v) = \nabla_v g(u, v) \cdot \nabla f(g(u, v)) \quad \forall (u, v) \in S,$$

where $\nabla_v \bar{h}(u, v)$ is the vector of partial derivatives of $\bar{h}$ with respect to the components of v. Using (A.3) and (A.5), we conclude that $\nabla_v \bar{h}(u, v) = 0$, for all $(u, v) \in S$. Since S is open and connected, it is easily shown that $\bar{h}$ is independent of v and there exists a continuously differentiable function $h : V \mapsto \Re$ such that

$$\bar{h}(u, v) = h(u) \quad \forall (u, v) \in S.$$

Here $V = F(R)$, which is obviously open and connected. For any $z \in R$, we have

$$f(z) = f(g(F(z), \Pi(z))) = \bar{h}(F(z), \Pi(z)) = h(F(z)),$$

as desired. $\square$

THEOREM A.2. *Let $F : Q \mapsto \Re^s$ be continuously differentiable, where $Q \subset \Re^r$ is open. We assume that* rank $(\nabla F(z)) < s$, *for all $z \in Q$, and that $\nabla F_1(z)$ (the first component mapping of F) is not equal to zero on the set Q. Then, there exists some positive integer i and some continuously differentiable function h such that*

$$F_{i+1}(z) = h(F_1(z), \cdots, F_i(z)) \quad \forall z \in R,$$

where R is some nonempty open subset of Q and F_i denotes the ith component mapping of F.

Proof. We let i be the largest index such that there exists some $\hat{z} \in Q$ with the property

$$\dim \operatorname{span} \{\nabla F_1(\hat{z}), \cdots, \nabla F_i(\hat{z})\} = i.$$

Clearly, $1 \leqq i < s$. By continuity, there exists some open subset $\hat{Q}$ of Q containing $\hat{z}$ such that $\nabla F_1(z), \cdots, \nabla F_i(z)$ are linearly independent for all $z \in \hat{Q}$. By our choice of the index i, we have

$$\nabla F_{i+1}(z) \in \operatorname{span} \{\nabla F_1(z), \cdots, \nabla F_i(z)\} \quad \forall z \in \hat{Q}.$$

By Theorem A.1, we see that there exists a continuously differentiable function $h : U \mapsto \Re$ such that

$$F_{i+1}(z) = h(F_1(z), \cdots, F_i(z)) \quad \forall z \in R$$

where R is some open subset of $\hat{Q}$ and $U = F(R)$. $\square$

THEOREM A.3. *Let $F: U \times V \mapsto \Re^s$ be a continuously differentiable mapping, where U and V are open subsets of $\Re^r$ and $\Re^t$, respectively. Let $(u^*, v^*) \in U \times V$ be such that* $\operatorname{rank}[\nabla_u F(u^*, v^*)] = s$ *and $F(u^*, v^*) = 0$. Then, there exists some nonempty open subsets $W \subset U$, $\bar{V} \subset V$ such that $u^* \in W$, $v^* \in \bar{V}$, and*

$$\{u \mid F(u, v) = 0\} \cap W$$

is nonempty and connected for all $v \in \bar{V}$. Furthermore, if $\{v_i \in V;\ i = 1, 2, \cdots\}$ is a sequence of vectors such that $\lim_{i \to \infty} v_i = v$ and $v \in \bar{V}$, then there exists a sequence $\{u_i \in W\}$ such that $F(u_i, v_i) = 0$ and $\lim_{i \to \infty} u_i = u$ for some $u \in W$.

Proof. We are in a situation where the assumptions of Lemma A.1 hold.[3] Let q, g, R, S be given as in Lemma A.1. Thus $(u, v) = g(q(u, v)) = g(F(u, v), \Pi(u), v)$, for all $(u, v) \in R$. Let g_u, g_v be the corresponding component mappings of g such that $u = g_u(q(u, v))$ and $v = g_v(q(u, v))$. Since S is open, we can take a connected open subset of S with the form $W_1 \times W_2 \times \bar{V}$ such that $W_1 \subset \Re^s$, $W_2 \subset \Re^{r-s}$, and $q(u^*, v^*) \in W_1 \times W_2 \times \bar{V}$. It is easy to check that W_2 is nonempty and connected and that $v^* \in \bar{V}$. Since g is a diffeomorphism, it follows that the set $g(W_1 \times W_2 \times \bar{V})$ is open. Moreover, we claim that g has following properties:

(a) $g_v(w_1, w_2, v) = v$ for all $(w_1, w_2, v) \in W_1 \times W_2 \times \bar{V}$;

(b) $\Pi(g_u(w_1, w_2, v)) = w_2$ for all $(w_1, w_2, v) \in W_1 \times W_2 \times \bar{V}$.

To prove the first property, let us write $(w_1, w_2, v) = q(u, v')$ for some $(u, v') \in R$. This is possible since $(w_1, w_2, v) \in S$. Hence, $(w_1, w_2, v) = (F(u, v'), \Pi(u), v')$. It follows that $v = v'$ and $(w_1, w_2, v) = q(u, v)$. Thus, $g_v(w_1, w_2, v) = g_v(q(u, v)) = v$, which proves (a). We now show the second property. As we have just seen, there exists some u such that $(w_1, w_2, v) = q(u, v)$ and $(u, v) \in R$. Thus, $(w_1, w_2, v) = (F(u, v), \Pi(u), v)$, from which it follows that $w_2 = \Pi(u)$. On the other hand, we have

$$\Pi(g_u(w_1, w_2, v)) = \Pi(g_u(q(u, v))) = \Pi(u),$$

from which it follows that $w_2 = \Pi(g_u(w_1, w_2, v))$.

Now let $W = g_u(W_1 \times W_2 \times \bar{V})$ and $S_u(v) = \{u \in U \mid F(u, v) = 0\}$. Since W is the projection of the open set $g(W_1 \times W_2 \times \bar{V})$, it follows that W is open in $\Re^r$. Also, it can easily be seen that $W \subset U$ and $u^* \in W$. Furthermore, we claim that

$$S_u(v) \cap W = \{g_u(0, w_2, v) \mid w_2 \in W_2\} \quad \forall v \in \bar{V}. \tag{A.7}$$

In fact, let us fix some $v \in \bar{V}$ and let $E(v)$ be the set in the right-hand side of (A.7). We will show that $E(v) \subset S_u(v) \cap W$. Clearly, $E(v) \subset W$. Thus, we only need to show that $E(v) \subset S_u(v)$. Let u be an element of $E(v)$. Then, there exists some $w_2 \in W_2$ such that $u = g_u(0, w_2, v)$. Since $q(u^*, v^*) = (F(u^*, v^*), \Pi(u^*), v^*) = (0, \Pi(u^*), v^*)$ and $q(u^*, v^*) \in W_1 \times W_2 \times \bar{V}$, we see that $0 \in W_1$. Thus, $(0, w_2, v) \in W_1 \times W_2 \times \bar{V}$. In light of property (a), we see that $v = g_v(0, w_2, v)$. Consequently,

$$F(u, v) = F(g_u(0, w_2, v), g_v(0, w_2, v)) = F(g(0, w_2, v)) = 0.$$

It follows that $E(v) \subset S_u(v) \cap W$.

For the reverse inclusion, given any $u \in S_u(v) \cap W$, we have $F(u, v) = 0$. Furthermore, there exists some $(w_1, w_2, v') \in W_1 \times W_2 \times \bar{V}$ such that $u = g_u(w_1, w_2, v')$. By property (b), we see that $\Pi(u) = w_2$. Thus $(0, w_2, v) = (F(u, v), \Pi(u), v) = q(u, v)$. Hence, $u = g_u(q(u, v)) = g_u(0, w_2, v)$. This implies that $u \in E(v)$, and (A.7) has been established. As a result, the set $S_u(v) \cap W$ is connected because, according to (A.7),

[3] Here we have assumed that $r > s$. The same argument works for the case $r = s$ except that Π should be dropped in the remaining proof.

it is the image of the connected set W_2 under a continuous mapping. Since $E(v)$ is nonempty for each $v \in \bar{V}$, (A.7) also shows that $S_u(v) \cap W$ is nonempty.

Given a sequence of vectors $\{v_i \in \bar{V};\ i = 1, 2, \cdots\}$ such that $\lim_{i\to\infty} v_i = v$ and $v \in \bar{V}$, let us pick $u_i = g_u(0, w_2, v_i)$, $i = 1, 2, \cdots$, where w_2 is some fixed vector in W. Hence, $u_i \in E(v_i)$ for all i. According to (A.7), we see that $F(u_i, v_i) = 0$. Furthermore, by the continuity of g_u, we see that

$$\lim_{i\to\infty} u_i = \lim_{i\to\infty} g_u(0, w_2, v_i) = g_u(0, w_2, v),$$

which is clearly in W. □

THEOREM A.4. *Let Q be an open set in $\Re^t$. Let also $F: Q \mapsto \Re^s$ be a continuously differentiable mapping such that*

$$\text{(A.8)} \qquad \operatorname{rank}(\nabla F(z)) = s \quad \forall z \in A,$$

where $A = \{z \mid F(z) = 0\}$. Suppose that $f: Q \mapsto \Re$ is continuously differentiable and is a constant function of z on A. Then,

$$\text{(A.9)} \qquad \nabla f(z) \in \operatorname{span}\{\nabla F(z)\} \quad \forall z \in A.$$

Proof. Consider the following constrained optimization problem:

$$\text{(A.10)} \qquad \min_{z\in A} f(z).$$

By assumption, each z in A is an optimal solution to (A.10). Since the regularity condition (A.8) ensures the existence of a set of Lagrange multipliers, the necessary condition for optimality gives the desired result ([L84, p. 300]). □

REFERENCES

[A80] H. ABELSON, *Lower bounds on information transfer in distributed computations*, J. Assoc. Comput. Mach., 27 (1980), pp. 384–392.

[AU83] A. V. AHO, J. D. ULLMAN, AND M. YANNAKAKIS, *On notions of information transfer in* VLSI *circuits*, in Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 133–139.

[BT89] D. P. BERTSEKAS, AND J. N. TSITSIKLIS, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[L84] D. G. LUENBERGER, *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1984.

[L89] Z. Q. LUO, *Communication complexity of some problems in distributed computation*, Ph.D. thesis, Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA, 1989.

[LT89] Z. Q. LUO AND J. N. TSITSIKLIS, *On the communication complexity of distributed algebraic computation*, Tech. Report LIDS-P-1851, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, *J. Assoc. Comput. Mach.*, submitted, 1989.

[MS82] K. MEHLHORN AND E. M. SCHMIDT, *Las Vegas is better than determinism in* VLSI *and distributed computing*, in Proc. 14th ACM *Symposium on Theory of Computing*, 1982, pp. 330–337.

[PE86] K. F. PANG AND A. EL GAMAL, *Communication complexity of computing the Hamming distance*, SIAM J. Comput., 15 (1986), pp. 932–947.

[PS82] C. H. PAPADIMITRIOU AND M. SIPSER, *Communication complexity*, in Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 196–200.

[PT82] C. H. PAPADIMITRIOU AND J. N. TSITSIKLIS, *On the complexity of designing distributed protocols*, Inform. and Control, 53 (1982), pp. 211–218.

[S65] M. SPIVAK, *Calculus on Manifolds*, W. A. Benjamin, New York, 1965.

[TL87] J. N. TSITSIKLIS AND Z. Q. LUO, *Communication complexity of convex optimization*, J. Complexity, 3 (1987), pp. 231–243.

[U84] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.

[Y79] A. C. YAO, *Some complexity questions related to distributed computing*, in Proc. 11th ACM Symposium on Theory of Computing, 1979, pp. 209–213.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 951-961, October 1991
© 1991 Society for Industrial and Applied Mathematics
009

THE SPECTRA OF INFINITE HYPERTREES*

JOEL FRIEDMAN†

Abstract. A model of regular infinite hypertrees is developed to mimic for hypergraphs what infinite trees do for graphs. Two notions of spectra, or "first eigenvalue," are then examined for the infinite tree, obtaining a precise value for the first notion and obtaining some estimates for the second. The results indicate agreement of the first eigenvalue of the infinite hypertree with the "second eigenvalue" of a random hypergraph of the same degree, to within logarithmic factors, at least for the first notion of first eigenvalue.

Key words. tree, graph, hypergraph, hypertree, second eigenvalue, eigenvalue, spectrum

AMS(MOS) subject classifications. primary, 05C50, 05C65, 68R10; secondary, 05C80, 68Q15

1. Introduction. In this paper we attempt to further the theory of the "second eigenvalue" of hypergraphs. The theory of the second eigenvalue of graphs is very rich, and can be used to give explicit constructions of graphs with certain geometric properties. However, its applications to problems, such as constructing *dispersers*, seem rather limited. This construction method generalizes naturally to hypergraphs and, in fact, constructing hypergraphs with small *second eigenvalue* can give much better dispersers (see [FW89]).

The problem with the notion of second eigenvalue for hypergraphs, as in [FW89], is that much of the eigenvalue theory for graphs does not generalize. For example, the "second eigenvalue" is not really an eigenvalue in any classical sense, and it is not clear that the known constructions of graphs with small second eigenvalue generalize in a strong way (e.g., to give better dispersers that can be given via graphs). In addition, there are various ways one can try to study the second eigenvalue of hypergraphs by relating them to the second eigenvalue of certain graphs, but the ones with which the author is familiar do not give, for example, better dispersers.

In graph theory, there is a strong connection between the second eigenvalue of a d-regular graph and the first eigenvalue of the infinite d-regular tree, its universal cover. In this paper we define for a uniform and regular hypergraph an infinite hypertree, and we analyze the "first eigenvalue" of the infinite hypertree. We do this for two notions of "first eigenvalue" or spectrum, but only for the first do we determine the precise answer. The analysis shows that, as with graphs, the second eigenvalue of random, regular hypergraphs is roughly the same as the first eigenvalue of the corresponding infinite hypertree; also, this value is roughly as small as one can get with any hypergraph of the same regularity.

The first notion of spectrum is the direct generalization of the definition in [FW89], but the second notion is new and perhaps has more structure to it. In particular, for every value of λ we define what it means to be spectral or nonspectral. In the first notion there is only a notion of what the "largest eigenvalue" (or "second largest" for a finite regular hypergraph) would be.

While the theorems proven here are fairly simple and do not directly imply facts about finite hypergraphs, the analysis does seem to show that there may be more ways to study "eigenvalues" of hypergraphs. Namely, there is a natural notion of universal

* Received by the editors October 15, 1990; accepted for publication January 4, 1991. This work was partially supported by a National Science Foundation Presidential Young Investigator grant CCR-8858788.
† Department of Computer Science, Princeton University, Princeton, New Jersey 08544. This paper was written while the author was visiting the International Computer Science Institute.

cover for hypergraphs, and its spectrum is, at least for superficial reasons, related to the spectrum of the finite hypergraphs it covers. In doing so, we introduce a new notion of spectrum, which may be worthy of study. The author hopes that the continued study of the spectra of hypergraphs will eventually yield explicit constructions of finite hypergraphs with small second eigenvalues.

In § 2 we discuss the relationship between the spectra of regular graphs and the corresponding infinite trees. In § 3 we define hypertrees and study their spectra. In § 4 we make some remarks about further directions of study.

2. The spectrum of graphs and infinite trees. In this section we summarize the connection between the spectrum of graphs and infinite trees, and we state two definitions of spectrum which can be generalized to hypergraphs: the one in [FW89], and one new one.

Let $G=(V, E)$ be a finite, undirected, d-regular graph, i.e., with every vertex having degree d, and let A be its adjacency matrix. Then A is an $n\times n$ matrix, $n=|V|$, which is symmetric and therefore has real eigenvalues $\lambda_1\leqq\lambda_2\leqq\cdots\leqq\lambda_n$. It is easy to see that $\lambda_1=d$ and that $\lambda_n\geqq -d$. There are examples of graphs (see [LPS86], [LPS88], [Mar84], [Mar87], [Mar88], [Hal86], [Chu88], and [Fri89]), for certain values of d and n, for which

$$\lambda_i\in[-2\sqrt{d-1}, 2\sqrt{d-1}] \quad \forall i\geqq 2, \tag{2.1}$$

and it is easy to show that for *fixed d* and $n\to\infty$, the interval in the above equation cannot be replaced by any smaller interval (independent of n; for d varying with n one can do better, as in [Mar84], [Mar87], and [Mar88]). It is also known that, for fixed d, "most" d-regular graphs on n vertices satisfy (2.1) if we enlarge the interval by an additive factor of $2\log d+C$ on each end, for some constant C, as $n\to\infty$ (see [Fri88]).

Next, let $T=(W, F)$ be the undirected, infinite, d-regular tree. We view its adjacency matrix as an operator B on $L^2(W)$, and it is easy to see that the spectrum of B is precisely the interval appearing in (2.1) (see [Car72] and the many references in [DK88]). This is not a coincidence, in that there is a standard technical reason for the similarity in the two, involving taking the trace of powers of A while viewing T as the universal cover of G. For example, the additional eigenvalue d which occurs in A's spectrum accounts for the fact that at the mth level of T (viewing T as rooted with root r), for large m, one expects roughly $1/n$ of the nodes to be the same V node as r; the derivation from this behavior is precisely related to the second eigenvalue of G (see, for example, [Fri88]).

Much of the theory used with graphs, such as forming matrices and considering their eigenvalues (via taking traces, etc.), seems to be difficult to generalize in a way that gives good results for the second eigenvalue of hypergraphs. We will give some definitions and propositions which generalize more directly for the infinite tree.

We begin with the standard calculation of the spectrum of the tree. We include the proof because it will be used in the hypergraph analysis.

PROPOSITION 2.1. *The spectrum of B, as above, is* $[-2\sqrt{d-1}, 2\sqrt{d-1}]$.

Proof. Fix a vertex v of T. Consider the "radial" function $f_r: T\to \mathbf{C}$ whose value at the mth level of T, i.e., at all vertices of distance m to v, is r^m. We have that

$$((B-\lambda I)f_r)(w)=\begin{cases} dr-\lambda & \text{if } w=v,\\ r^{m-1}((d-1)r^2-\lambda r+1) & \text{if } w \text{ lies on level } m\geqq 1.\end{cases}$$

Fix a λ with $|\lambda|>2\sqrt{d-1}$. There exists a solution, r, to

$$(d-1)r^2-\lambda r+1=0, \tag{2.2}$$

with $|r|<(d-1)^{-1/2}$, which makes the resulting f_r lie in $L^2(W)$. For such an r we have $dr-\lambda\neq 0$, and therefore the equation in x,

$$(B-\lambda I)(x)=\delta_v \tag{2.3}$$

has an $L^2(W)$ solution x, where δ_v is 1 on v and 0 elsewhere. Writing an arbitrary $w\in L^2(W)$ as a (possibly infinite) linear combination of such δ's and using linearity, we can solve the above equation in x with δ_v replaced by any w, with $\|x\|$ bounded by a constant times $\|w\|$. Therefore $B-\lambda I$ is invertible and λ lies outside the spectrum of B.

On the other hand, we claim that for $|\lambda|<2\sqrt{d-1}$, (2.3) has no solution $x\in L^2(W)$. Indeed, if such an x existed, then its symmetrization, $\tilde{x}$, whose value at each vertex on the mth level is the average of the mth level value of x, would also be an $L^2(W)$ solution of (2.3). Then the values $\tilde{x}_i$ of $\tilde{x}$ at the ith level satisfy

$$(d-1)\tilde{x}_{i+2}^2-\lambda\tilde{x}_{i+1}+\tilde{x}_i=0\quad\forall i\geqq 0,$$

and so

$$\tilde{x}_m=c_1r_1^m+c_2r_2^m$$

for some constants c_i and with r_i being the roots of (2.2). But for λ, the roots r_1, r_2 are both of absolute value $(d-1)^{1/2}$, contradicting the fact that $\tilde{x}\in L^2(W)$.

Also, $\lambda=\pm 2\sqrt{d-1}$ is in the spectrum, either by modifying the equation for $\tilde{x}_m$ in the above, or by noting that the spectrum is a closed set. Since B is selfadjoint, its spectrum is real; we have now determined its entire spectrum. □

Since B is selfadjoint, the above proposition implies the following.

COROLLARY 2.2. *For any $x, y\in L^2(W)$, $|(Bx,y)|\leqq 2\sqrt{d-1}\,\|x\|\,\|y\|$, and $2\sqrt{d-1}$ is the best constant possible. Equivalently, the $L^2(W)$ norm of B is $2\sqrt{d-1}$.*

We provide a simpler proof of this which immediately generalizes to hypertrees. The upper bound is, in a sense, related to "integration by parts" eigenvalue bounds suggested to the author by Sarnak.

Proof. It suffices to consider the case $\|x\|=\|y\|=1$. We have

$$(Bx,y)=\sum_{(i,j)\in F}x_iy_j,$$

where we think of F as containing one copy of (i,j) and one of (j,i) for every undirected edge $\{i,j\}$ it contains. For those terms x_iy_j in the above sum with i nearer to v than j, write

$$x_iy_j\leqq\tfrac{1}{2}(\sqrt{d-1}\,x_i^2+y_j^2/\sqrt{d-1});$$

for those with j nearer to v, reverse x_i and y_j. Remembering that every vertex except v has one neighbor closer to v and $d-1$ further from v (and that v has d neighbors, all further away from v), we get

$$|(Bx,y)|\leqq\sum_{w\in W-\{v\}}2\sqrt{d-1}(x_w^2+y_w^2)+\frac{d}{\sqrt{d-1}}(x_v^2+y_v^2)\leqq 2\sqrt{d-1}.$$

This provides an upper bound on the norm of the bilinear form associated with B. For the lower bound, consider for small $\varepsilon>0$ the radial function, f_r, with $r=(d-1)^{-1/2}(1-\varepsilon)$. We have

$$\|f_r\|^2=1+\sum_{m=1}^{\infty}d(d-1)^{m-1}((d-1)^{-1/2}(1-\varepsilon))^{2m}=\frac{d}{d-1}\frac{1}{2\varepsilon}+O(1),$$

and, similarly,

$$(Bf_r, f_r) = \frac{2d}{\sqrt{d-1}} \frac{1}{2\varepsilon} + O(1),$$

so that taking $\varepsilon \to 0$ gives the desired lower bound. □

The above argument also shows that the norm of B is never achieved by any vector in $L^2(W)$. We now state the precise definitions which we intend to carry over to hypergraphs in the next section. In what follows, we take $L^2(W)$ to be the space of *complex-valued* functions, although in a lot of places it suffices to take real-valued functions.

DEFINITION 2.3. The *spectral radius* of the infinite tree T with adjacency matrix B is the $L^2(W)$ norm of the bilinear form (Bx, y).

DEFINITION 2.4. For the infinite tree T with adjacency matrix B, a number $\lambda \in \mathbf{C}$ is said to be *nonspectral* if

(i) fundamental solutions exist for $B - \lambda I$, i.e., there exists an x with $(B - \lambda I)x = \delta_v$,
(ii) for every $y \in L^2(W)$, $(B - \lambda I)x = y$ has a solution $x \in L^2(W)$,
(iii) the above x is uniquely determined,
(iv) the above x's norm is bounded by a constant times y's.

If any of the above fail to hold, λ is said to be spectral. Also, a $\lambda \in \mathbf{R}$ is said to be a *spectral upper bound* (respectively, lower bound), if it is nonspectral and

(i) for every real-valued (x, y) pair with $(B - \lambda I)x = y$, we have that (x, y) is nonnegative (respectively, nonpositive).

Note that in the present circumstances, the first condition of nonspectrality implies all the others.

3. The spectra of hypergraphs and infinite hypertrees. For simplicity, we will state all theorems in this section for 3-uniform hypergraphs, though all the theorems here easily generalize to t-uniform hypergraphs for any fixed t.

We review our terminology for hypergraphs; see [FW89] for details. A 3-uniform hypergraph is a collection $G = (V, E)$ of a set V and a collection of subsets of V, E, such that each subset $e \in E$ has size three. Out of this data we can form a trilinear form τ analogous to the bilinear form associated with the adjacency matrix of a graph, namely,

$$\tau(x, y, z) = \sum_{i,j,k \in V} x_i y_j z_k \tau_{ijk}$$

for $x, y, z \in L^2(V)$, where

$$\tau_{ijk} = \begin{cases} 1 & \text{if } \{i, j, k\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For our purposes it is simpler to think of a hypergraph as a trilinear form τ, with τ_{ijk} nonnegative integers. In [FW89], the *second eigenvalue* of τ is defined to be

$$\left\| \tau - \frac{d}{n} \mathscr{E} \right\|, \tag{3.1}$$

where $\mathscr{E}$ is the trilinear form with all $\mathscr{E}_{ijk} = 1$, $n = |V|$, $d = \sum_{ijk} \tau_{ijk}/n^2$, and the norm is the norm as a trilinear form on $L^2(V)$, i.e.,

$$\|\sigma\| = \max_{\|x\| = \|y\| = \|z\| = 1} |\sigma(x, y, z)|. \tag{3.2}$$

It is shown there that for a "randomly chosen" τ on n vertices with dn^2 hyperedges, $d > C \log n$, the second eigenvalue of τ is, with high probability, roughly $\sqrt{d}$, to within a factor of $C(\log n)^{3/2}$. This can be compared with its "first eigenvalue," namely, the norm of τ, which is roughly $d\sqrt{n}$. One can give explicit examples of hypergraphs with second eigenvalue around $d^{1/2}n^{1/4}$, but this does not give improvements for the dispenser construction. However, any explicit construction of hypergraphs with smaller exponents would yield improvements.

First we form a notion of the infinite hypertree. Before doing so, notice that the first and second "eigenvalues" of a random τ (as above) are not powers of each other. We can remedy this by working with the $L^3(V)$ norm. Indeed, as remarked in [FW89], all the theorems generalize to L^p for any p (using (3.1), but taking $\|\cdot\|$ on x, y, z to be the $L^p(W)$ norm in (3.2)), and choosing $p = 3$ gives first eigenvalue (i.e., norm of τ) to be roughly dn and second eigenvalue to be roughly $(dn)^{1/3}$. While the L^3 norm may seem strange, it has other advantages. For one, defining eigenvalues in terms of bilinear forms involves picking a fixed inner product on the space in question. A natural analogous trilinear product is

$$\mathscr{I}(x, y, z) = \sum_{i \in V} x_i y_i z_i,$$

and when using $\mathscr{I}$, it seems natural to work with the L^3 norm. Also, use of the L^3 norm suggests that we work with $k = dn$ as the "degree" of our hypergraph, and this notion of degree gives a good model of a universal cover.

To define our hypertree, fix a value of k. Start by taking one triangle, and to each of its vertices glue $k-1$ triangles, all disjoint except that they meet in the one vertex. For each newly created vertex, create $k-1$ new triangles. The resulting infinite hypergraph, $T = (W, F)$, is depicted in Fig. 1. On T we have a notion of distance, defining two vertices to be neighbors (i.e., distance 1) if they both lie in some triangle. Thus, if the top vertex of Fig. 1 is v, then all the vertices of distance 1 to v lie in the row of vertices directly below v, those of distance two in the next row, etc.

We call the above hypergraph T the *k-regular hypertree*. Why do we use this model? Aside from the fact that the first and second L^3 eigenvalues of a random hypergraph are roughly powers of k, this hypertree is, in a natural way, the universal cover of any hypertree of *degree* k, i.e., in which each vertex is incident on exactly k (hyper)edges. More precisely, taking a *morphism* of hypergraphs to be a map of vertices

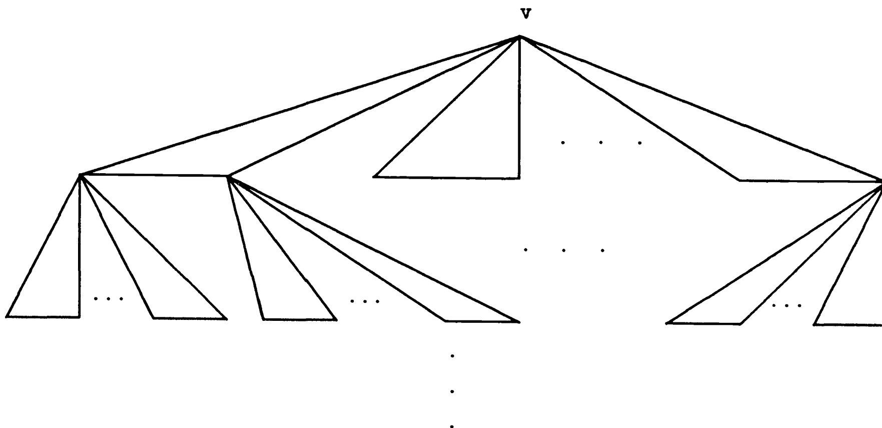


FIG. 1. *The infinite hypertree.*

which maps edges to edges, and a *cover* to be a locally invertible morphism, our k-regular tree is precisely a (the) universal cover. In particular, any finite hypergraph of degree k is isomorphic to a quotient of our hypertree modulo some equivalence relation on the vertices. We could have constructed the hypertree by growing triangles off of edges, requiring that each edge meet d triangles; this would be closer to the definition of regularity appearing in [FW89], but seems to be harder to work with.

The infinite hypertree T has a trilinear form τ associated to it via the above procedure, in which each triangle gives rise to six 1's in τ. We expect, in analogy with the infinite tree, that the range of the L^3 "spectrum" of τ, appropriately defined, should be roughly $k^{1/3}$, i.e., what the second eigenvalue is for a random hypergraph. Similar to the "eigenvalue" definitions in [FW89], we make the following definition.

DEFINITION 3.1. The *spectral radius* of a trilinear form is its L^3 norm.

PROPOSITION 3.2. *The spectral radius of* τ, *i.e., its* $L^3(W)$ *norm, is* $3(2k-2)^{1/3}$.

Proof. Fix a vertex v in T. Each triangle is of the form $\{i, j, l\}$, with i a vertex of distance m to v, and j, l of distance $m+1$ to v for some m. To estimate $\tau(x, y, z)$ for vectors x, y, z of norm 1, we can estimate, for any positive γ,

$$|x_i y_j z_l| \leqq \tfrac{1}{3}(\gamma^{-2/3}|x_i|^3 + \gamma^{1/3}|y_j|^3 + \gamma^{1/3}|z_l|^3),$$

and similarly for the other five terms arising from $\{i, j, l\}$. Then, summing as before yields

$$|\tau(x, y, z)| \leqq \max\,(2\gamma^{1/3} + 2(k-1)\gamma^{-2/3}, 2k\gamma^{-2/3}),$$

the first term in the max accounting for the contribution of vertices in $W-\{v\}$, the second for v's (i.e., the components of x, y, z at these vertices). Taking $\gamma = (2k-2)^{1/3}$ yields the desired upper bound on $\|\tau\|$.

On the other hand, let f_r be the function on W whose value at each vertex distance m to v is r^m. Then for small $\varepsilon > 0$ and $r = (2k-2)^{-1/3}(1-\varepsilon)$, we have

$$\sum_{w \in W} f_r^3(w) = \frac{k}{k-1}\,\frac{1}{3\varepsilon} + O(1),$$

and

$$\tau(f_r, f_r, f_r) = 6k(2(k-1))^{-2/3}\,\frac{1}{3\varepsilon} + O(1),$$

so that

$$\|\tau\|_{L^3(W)} = 6 \cdot 2^{-2/3}(k-1)^{1/3} = 3(2k-2)^{1/3}. \qquad \square$$

Next we consider the analog of Definition 2.4. The equation $(B - \lambda I)x = y$ can be written as

$$\mathscr{B}(x, w) - \lambda(x, w) = (y, w) \quad \forall w \in L^2(W), \tag{3.3}$$

where $\mathscr{B}(x, w) = (Bx, w)$ is the bilinear form associated with B. To generalize this to hypergraphs, note that we have a natural trilinear form τ to replace $\mathscr{B}$, and the question becomes what to use for the standard inner product $(\cdot, \cdot)$. For the latter, we suggest using the above-defined $\mathscr{I}$. The trilinear form $\mathscr{I}$ seems the most natural to use, although it does have all the nice properties of the standard bilinear inner product, for example, $\mathscr{I}(u, u, u)$ is not generally equal to $\|u\|^3$. However, using $\mathscr{I}$ does seem related to our previous notion of spectral radius.

The question now is how many fixed variables, such as x or y, to use in an analog of (3.4), and how many test variables, such as w, to use and where to place them. In this paper we investigate the equation

$$\tau(x, x, u) - \lambda\mathscr{I}(x, x, u) = \mathscr{I}(y, y, u) \quad \forall u \in L^3(W). \tag{3.4}$$

Arguably, we should replace x and/or y by two variables, but we leave it in this form for simplicity and recall that if τ is symmetric, then to find its norm is suffices to check $\tau(x, y, z)$ for $x=y=z$.

We pause to make two remarks about $\mathscr{I}$. First of all, for any $x, y, z \in L^3(W)$,

$$|\mathscr{I}(x, y, z)| \leqq \|x\|\,\|y\|\,\|z\|;$$

this follows from two applications of Hölder's inequality or from estimating as in the proof of Proposition 3.2. Second, for any $x \in L^3(W)$ we use the notation x' to denote the vector given by $|x'_w| = |x_w|$ and $x'_w x_w^2 = |x_w|^3$. Thus x' differs pointwise from x by numbers of absolute value 1, has the same norm as x, and satisfies

$$\mathscr{I}(x, x, x') = \|x\|^3.$$

Definition 3.3. For a trilinear form τ a number $\lambda \in \mathbf{C}$ is said to be *nonspectral* if

(i) Fundamental solutions exist for $\tau - \lambda\mathscr{I}$, i.e., there exists a solution x for (3.4) with $y = \delta_v$,

(ii) for every $y \in L^3(W)$, (3.4) has a solution $x \in L^3(W)$,

(iii) any solution x, for (ii), has its norm bounded by a constant times y's.

If any of the above fail to hold, λ is said to be spectral. Also, a $\lambda \in \mathbf{R}$ is said to be a *spectral upper bound* (respectively, lower bound), if it is nonspectral and

(i) for every pair x, y satisfying (3.4) and with x real and each y_w either real or purely imaginary, we have that $\mathscr{I}(y, y, x')$ is nonnegative (respectively, nonpositive).

We have omitted the condition that x is uniquely determined, since this will never be the case (see below). As before, L^3 refers to complex-valued functions; it is not clear that real L^3 works as well here (for λ real). In the definition of spectral upper and lower bounds, we have allowed some of y's values to be purely imaginary to make sure that every real x has a corresponding y (see (3.5) below).

We now state the main result of the paper, whose proof comprises the rest of this section.

Theorem 3.4. *Any $\lambda \in \mathbf{C}$ with absolute value bigger than the spectral radius is nonspectral. In particular, any such positive* (*respectively, negative*) *λ is a spectral upper* (*lower*) *bound.*

To begin the analysis, note that (3.4) is equivalent to requiring that for all $w \in W$,

$$\sum_{j=1}^{k} x_{\alpha(w,j)} x_{\beta(w,j)} - \lambda x_w^2 = y_w^2, \tag{3.5}$$

where each $\alpha(\cdot, l)$ and each $\beta(\cdot, l)$ is a permutation of W.

We will first discuss in detail the situation for real λ, and remark later about complex λ.

We claim that almost every real λ has a complex-valued fundamental solution. That is, letting f_r be as before, we see from (3.5) that f_r will be a fundamental solution if and only if

$$(k-1)r^3 - \lambda r + 1 = 0.$$

Proposition 3.5. *The equations $s^3 - \alpha s + 1 = 0$ has*

(i) *all roots of absolute value* 1 *if* $\alpha = 0, 2$;

(ii) *one positive real root of absolute value less than* 1, *two complex roots of absolute value greater than* 1 *if* $\alpha < 0$;

(iii) *one negative real root of absolute value less than* 1, *two real roots of absolute value greater than* 1 *if* $\alpha > 2$;

(iv) *one real root of absolute value greater than* 1, *two complex roots of absolute value less than* 1 *if* $0<\alpha<(27/4)^{1/3}$;

(v) *one real root of absolute value greater than* 1, *two real roots of absolute value less than* 1 *if* $(27/4)^{1/3}\geqq\alpha<2$ (*the latter two roots being a double root when equality holds*).

Proof. The proof follows easily from the fact that the above equation has discriminant $4\alpha^3-27$ and that $g(s)=s^3-\alpha s+1$ has $g(-1)-\alpha$ and $g(1)=2-\alpha$.

From this proposition it follows that there is always a radial fundamental solution, $f_r\in L^3(W)$, except for $r=0, 2(k-1)^{1/3}$. The existence of a solution to (3.4) does not follow, because of the nonlinearity. However, assuming that λ is larger than the spectral radius, one can solve (3.4) for any finitely supported y (i.e., which is zero at all but a finite number of vertices), and pass to the limit for general y. In proving both steps we will use the estimate in the (direct) proof of Corollary 2.2, and we do not know what happens in general.

THEOREM 3.6. *Let* $|\lambda|>3(2k-2)^{1/3}$. *Then for any finitely supported* y, *there exists a solution* x *to* (3.4).

Proof. As usual, fix a vertex v. Suppose y is supported on the set of vertices of distance less than or equal to m to v. Consider the class of vectors x whose values at the vertices of distance less than or equal to m is arbitrary, and whose values at each vertex of distance greater than or equal to $m+1$ is given as r times the value of its neighbor that is closest to v. Such x are "eventually radial"; we have depicted the case $m=1$ in Fig. 2. For any such x it is clear that (3.5) holds for any w of distance greater than or equal to $m+1$ from v. To satisfy this equation at the other w's, we get N quadratic equations in the N variables x_w where w ranges over the vertices on levels less than or equal to m. It follows that the system of equations has at least one solution in x over N-dimensional (complex) projective space.[1] But it is easy to check that all

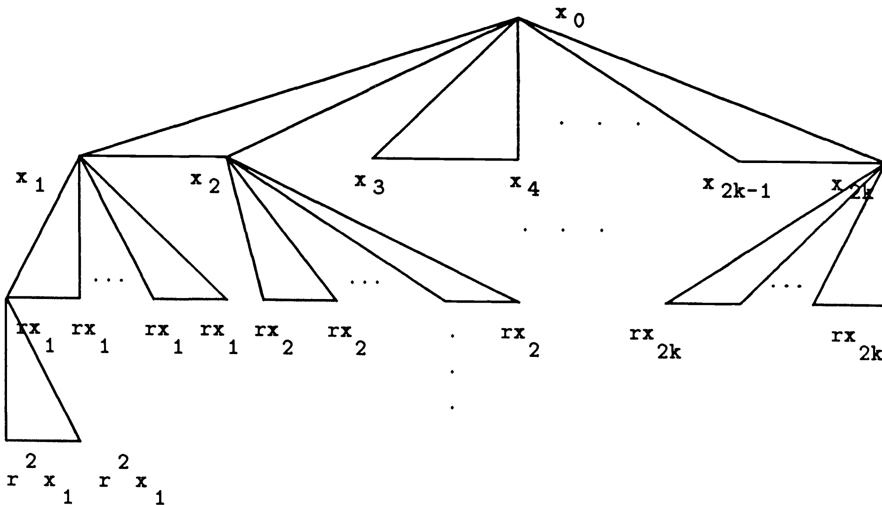


FIG. 2. *A general solution* x *for* $m=1$.

[1] See, for example, [Har77, I.7.2.] The reader who is unfamiliar with algebraic geometry can recall that if an $N\times N$ linear system $Ax=0$ has no nontrivial solutions, x, then for any b the system $Ax=b$ has a solution. This turns out to be true if the linear equations are replaced by any homogeneous equations, assuming the underlying field is algebraically closed. This is the point of making the calculation which follows.

solutions of these N equations lie in affine space. This is because a solution in projective space is precisely a nontrivial solution x of the same system with the y_i's replaced by zeros. But this would imply

$$\tau(x, x, u) - \lambda \mathcal{I}(x, x, u) = 0 \quad \forall u \in L^3(W).$$

In particular, choosing $x' \in L^3(W)$ given by $|x'_v| = |x_v|$ and $x'_v x_v^2 = |x_v|^3$, we get

$$|\lambda| \|x\|^3 = |\lambda \mathcal{I}(x, x, x')| = |\tau(x, x, x')| \leqq 3(2k-2)^{1/3} \|x\|^3,$$

which contradicts the nontriviality of x. Hence all projective solutions of the aforementioned system lie in affine space, and so there exists at least one such solution. □

We mention, as in the footnote, that the solvability of (3.4) for arbitrary y is related, in a certain sense, to the nonexistence of nontrivial solutions for the special case $y = 0$. We will return to this point in the next section.

THEOREM 3.7. *Let $|\lambda|$ exceed the spectral radius. Then, for any $y \in L^3(W)$, there exists a solution $x \in L^3(W)$ to (3.4). Furthermore, for any such solution x, we have $\|x\| \leqq C\|y\|$, where C depends only on $|\lambda|$.*

Proof. A priori, for any solution x as above, we have

$$\tau(x, x, x') - \lambda \mathcal{I}(x, x, x') = \mathcal{I}(y, y, x'),$$

so that

$$(|\lambda| - 3(2k-2)^{1/3}) \|x\|^3 \leqq |\mathcal{I}(y, y, x')| \leqq \|y\|^2 \|x\|.$$

Hence

$$\|x\| \leqq C\|y\| \quad \text{for } C = (|\lambda| - 3(2k-2)^{1/3})^{-1/2}.$$

Now fix $y \in L^3(W)$, and let y^n be a sequence of finite truncations of y converging to y, i.e., y^n are finitely supported, y^n_w is either 0 or y_w for each n and w, and $\|y^n - y\| \to 0$ as $n \to \infty$. For each n choose a solution x^n to (3.4) for y^n. Since $\|y^n\|$ is bounded, so is $\|x^n\|$, and any weakly converging subsequence of the x^n's gives us a solution, x, to (3.4).

In more detail, for each w, the sequence x^n_w is bounded, and so we can assume, by passing to a subsequence, that for each $w \in W$ we have $x^n_w \to x_w$ for some $x_w \in \mathbf{C}$ (since W is countable, using a "diagonal subsequence"). A standard argument shows that the resulting vector x lies in $L^3(W)$: let $\tilde{x}$ be any truncation of x. Then $\tilde{x} \in L^3(W)$, and

$$\mathcal{I}(\tilde{x}, \tilde{x}', x^n) \leqq \|\tilde{x}\|^2 \|x^n\|.$$

Taking any subsequence of n's tending to infinity, the left-hand-side converges to $\|\tilde{x}\|^3$, and so

$$\|\tilde{x}\| \leqq \liminf \|x^n\|.$$

Since this holds for any truncation $\tilde{x}$ of x, this holds for x itself. Finally, for each w, (3.5) holds for x and y, since for all sufficiently large n (depending on w), the equation holds for x^n and y, and this equation involves only terms x^n_v with v ranging over a finite set. Hence x is a solution to (3.4), and it clearly satisfies the a priori bound given. □

To complete the proof of Theorem 3.4, it is clear that in general any real positive (respectively, negative) λ which is spectral and which exceeds the spectral radius must be a spectral upper (lower) bound. Finally, the entire discussion of nonspectrality goes

through for $\lambda \in \mathbf{C}$ of absolute value exceeding the spectral radius, with minor modifications. We start by observing that the equation

$$s^3 - \alpha s + 1 = 0$$

can only have all roots s_1, s_2, s_3 of absolute value less than or equal to 1 if

$$|\alpha| = |s_1 s_2 + s_1 s_3 + s_2 + s_3| \leqq 3.$$

This implies that any complex λ with $|\lambda| > 3(k-1)^{1/3}$ has a fundamental solution. The rest of the analysis goes through virtually word for word, to show that λ's with $|\lambda| > 3(2k-2)^{1/3}$ are nonspectral. □

We remark that we never have uniqueness in (3.4). One obvious reason is that if x is a solution, then so is $-x$. However, this is not the only source of nonuniqueness. For example, if the support of $x \in L^3(W)$ is any set of minimum distance 3, i.e., the distance between any two distinct vertices is at least 3, then there is a $y \in L^3(W)$ which satisfies (3.4), and any pattern of sign changes in x yields another solution. So, in general, there can be an infinite number of solutions x for a given y.

4. Finite versus infinite hypergraphs. We finish with some remarks on the question of constructing finite hypergraphs with small second eigenvalues.

The basic question is to construct hypergraphs on n nodes with dn^2 edges whose second eigenvalue, measured in L^3, is roughly $(dn)^{1/3}$. For the application to dispersers, it would be desirable that the hypergraph be constructable in polylogarithmic time in n and d. So, for example, in the analogous construction problem for graphs, the graphs given in [LPS86] and [Mar84] are not known to be constructible so quickly, but those of [Hal86], [Chu88], and [Fri89] are. Any construction yielding a hypergraph of second L^3 eigenvalue less than or equal to $(dn)^\beta$ for some $\beta < \frac{1}{2}$ would improve the best disperser construction known at present; for example, it could boost ε-weak sources for some $\varepsilon < \frac{1}{2}$.

Clearly, any symmetric, regular hypergraph can be represented as the quotient of an infinite tree whose vertices are identified in some way. The question becomes, then, are there concise properties of the universal covers which control the second eigenvalue of a finite hypergraph?

We can suggest the following question, which is even interesting for graphs. Given a finite graph and an eigenvector, is there a direct way to prove that its eigenvalue is small by associating to it some vector, or perhaps probability space of vectors, on its universal cover? This association should work for all vectors perpendicular to $e = (1, 1, \cdots, 1)$ but not for e itself, and would have to involve some properties of the map from the universal cover to the graph (since no good bound holds for all finite graphs). This is a suitably vague question, but the intention is to develop methods that could carry over to hypergraphs to yield better second eigenvalue bounds.

For finite hypergraphs, one may be able to define notions of spectrality as in Definition 3.3, but one would probably want to modify the definition. For example, consider the following consequence of the standard variational argument.

PROPOSITION 4.1. *Let τ be a symmetric, trilinear form on $L^3(V)$, where V is a finite set, and let E be a linear subspace of $L^3(V)$. If $|\tau(v, v, v)|$ over unit vectors $v \in E$ is maximized at $v = x$, then there is a λ' such that*

$$\tau(x, x, u) - \lambda' \mathcal{I}(x, x, u) = 0 \quad \forall u \in E.$$

Proof. If $\mathcal{I}(x, x, u) = 0$, it follows that $\tau(x, x, u) = 0$ by considering $|\tau(v, v, v)|$ with $v = x + \varepsilon u$ and ε small. So choosing λ' to make the above equation hold for any particular u with $\mathcal{I}(x, x, u) \neq 0$ will work. □

The point here is that for λ' to equal λ it is necessary that $|\mathcal{I}(x, x, x)| = \|x\|^3$, which will not generally be the case. A solution to the above equation is clearly related to the solvability of (3.4) for $\lambda = \lambda'$ (i.e., too many solutions to the above prohibits a solution to (3.4)), but we would not expect λ and λ' to agree over, for example, the subspace E of u's with $\mathcal{I}(\vec{1}, \vec{1}, u) = 0$, where $\vec{1}$ is the all-1's vector. Hence, to study an analog of Definition 3.3 for the second eigenvalue of finite hypergraphs, we would expect some modification.

We remark that all the theorems stated in § 3 generalize easily to *t-uniform* hypergraphs $G = (V, E)$, i.e., where each $e \in E$ is a subset of size t. The resulting eigenvalue for the k-regular, t-uniform hypertree is $(k-1)^{1/t} t!(t-1)^{(1-t)/t}$, which for fixed k is within polylog factors in k of the right answer, i.e., of the lower bound and of the upper bound for random hypergraphs.

Acknowledgments. The author wishes to thank Frederic Bien, Peter Sarnak, and Eli Shamir for useful conversations.

REFERENCES

[Car72] P. CARTIER, *Fonctions harmoniques sur un arbre*, Symposia Math., 9 (1972), pp. 203–270.

[Chu88] F. R. K. CHUNG, *Diameters and eigenvalues*, preprint, 1988.

[DK88] J. DODZIUK AND L. KARP, *Spectral and function theory for combinatorial Laplacians*, in Geometry of Random Motion, American Mathematical Society, Providence, RI, 1988, pp. 25–40.

[Fri88] J. FRIEDMAN, *On the second eigenvalue of random d-regular graphs*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, August 1988; Combinatorica, to appear.

[Fri89] ———, *Some graphs with small second eigenvalue*, Tech. Report, Department of Computer Science Princeton University, October 1989.

[FW89] J. FRIEDMAN AND A. WIGDERSON, *On the second eigenvalue of hypergraphs*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, September 1989.

[Hal86] M. HALL, *Combinatorial Theory*, Second Edition, John Wiley & Sons, 1986.

[Har77] ROBIN HARTSHORE, *Algebraic Geometry*, Springer-Verlag, Berlin, 1977.

[LPS86] A. LUBOTZKY, R. PHILLIPS, AND P. SARNAK, *Explicit expanders and the Ramanujan conjectures*, in 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 204–246.

[LPS88] A. LUBOTZKY, R. PHILLIPS, AND P. SARNAK, *Ramanujan graphs*, Combinatorica, 8 (1988), pp. 261–277.

[MAR84] G. A. MARGULIS, *Arithmetic groups and graphs without short cycles*, in Sixth Internat. Symposium on Information Theory, Tashkent, U.S.S.R., Abstracts, Vol. I, 1984, pp. 123–125. (In Russian.)

[Mar87] ———, *Some new constructions of low-density parity check codes*, in Third Internat. Symposium on Information Theory, Convolution Codes and Multi-user Communication, Sochi, 1987, pp. 275–279. (In Russian.)

[Mar88] ———*Explicit group theoretic constructions of combinatorial schemes and their applications for the constructions of expanders and concentrators*, J. Prob. Inform. Trans., 24 (1988), pp. 39–46.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 962–986, October 1991
©1991 Society for Industrial and Applied Mathematics
010

ON THE COMPLEXITY OF LEARNING MINIMUM TIME-BOUNDED TURING MACHINES*

KER-I KO†

Abstract. The following problems about time-bounded program-size complexity are studied: (1) for a given string x, a size bound s, and a time bound t, whether there exists a Turing machine of size less than or equal to s that prints x in t moves; (2) for two given finite sets Y and Z of strings, a size bound s, and a time bound t, whether there exists a Turing machine of size less than or equal to s that operates in time t and accepts all $y \in Y$ and rejects all $z \in Z$. These problems are fundamental in complexity theory and feasible learning theory. The complexity of these problems is apparently between P and NP, but appears very difficult to classify precisely. These problems are attacked by the approach of relativization. It is shown that for certain variations of the problems, they could be either polynomial-time computable or not polynomial-time computable, depending on different oracles. Furthermore, there are oracles relative to which they are not complete for NP under the polynomial-time Turing reductions, but are complete for NP under the strong NP reductions.

Key words. learning, polynomial time, Turing machines, generalized Kolmogorov complexity, NP-completeness

AMS(MOS) subject classification. 68C25

1. Introduction. Time-bounded program-size complexity (generalized Kolmogorov complexity, or simply KT-complexity) has recently found many interesting applications in complexity theory (see, for instance, Hartmanis [8]; Sipser [23]; Balcázar and Book [3]; Longpré [20]; Ko, Orponen, Schöning, and Watanabe [14]), as well as the theory of pseudorandomness (Levin [16], Ko [11], Huynh [9]). Other applications to lower bound proof techniques and cryptography are also known (see Li and Vitanyi [17] for a survey).

While these applications are very interesting, some fundamental questions about time-bounded program-size complexity remain unanswered. Consider the following problem.

> *MINKT*: for a given string x, a given size bound s, and a given time bound t, determine whether there exists a Turing machine T of size less than or equal to s that prints x within t moves.

The complexity of this problem is fundamental to a number of applications in complexity theory. For instance, this problem is critical to the question of whether the Martin–Löf characterization of randomness by Kolmogorov complexity holds for the polynomial-time version [11]. Vazirani and Vazirani [26] studied the complexity of the minimum-size Turing machine problem in the form of real-time transducers.

It is obvious that this problem is in NP if the time bound t is given in the unary form. Intuitively it is very unlikely to be in P. In fact, we know of no polynomial-time heuristics to either attack any of its subproblems or to approximate it in any sense. The only applicable technique seems to be the exhaustive search. And yet this is also unlikely to be complete for NP, because it does not seem to possess the structural properties, such as paddability and self-reducibility, that are shared by most other

*Received by the editors April 18, 1990; accepted for publication (in revised form) November 29, 1990. This research was supported in part by National Science Foundation grant CCR-8801575.

†Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794.

known NP-complete problems (see, for instance, Balcázar, Diaz, and Gabarró [4]). The precise classification of its complexity thus appears to be a challenging problem in complexity theory.

The close connection between program-size complexity and inductive inference has been known through the work of Solomonov [24] and was more recently surveyed by Li and Vitanyi [18]. A fundamental problem in the theory of polynomial-time inductive inference, or the feasible learning theory, is to identify from a given set S of input/output pairs, the concept C in the prespecified domain that is *consistent* with the input/output pairs in S. For instance, Blumer, Ehrenfeucht, Haussler, and Warmuth [5] have shown that Valiant's distribution-free probabilistic learning model [25] for a big class of learning problems is essentially equivalent to the consistency problem. Applying the principle of Occam's Razor, one is often interested in the minimum-size concept C that is consistent with S. The consistency problem of some specific domains of concepts has been well studied. For instance, the consistency problem of boolean formulas and boolean circuits has been studied in Kearns, Li, Pitt, and Valiant [10]; the consistency problem of minimum-state finite automata is shown to be NP-complete in Gold [7] and Pitt and Warmuth [21]; and the consistency problem of maximum-length pattern languages has been studied in Angluin [2] and Ko, Marron, and Tzeng [13]. However, very little work has been done on the consistency problem for the more general concepts of time-bounded Turing machines. We formulate the problem as follows.

> *MINLT*: for two given finite sets Y and Z of strings, a given size bound s, and a given time bound t, determine whether there exists a Turing machine T of size less than or equal to s such that for each $y \in Y$, $T(y)$ accepts in t moves, and for each $z \in Z$, $T(z)$ rejects in t moves.

Although problem *MINLT* seems to have a different motivation from problem *MINKT*, the two problems can be seen to be quite similar if we regard problem *MINKT* as a simpler version of the learning problem *MINLT*, i.e., learning the Turing machine program from one of its outputs. In addition, the two problems have similar complexity-theoretic properties. Namely, if the size bound s and the time bound t are given in the unary form, then *MINLT* is in NP and is not likely to be polynomial-time solvable. Also, it does not seem to possess the structural properties that characterize known NP-complete problems. The precise classification of its complexity is of interest in feasible learning theory.

While we are tempted to conjecture that *MINKT* and *MINLT* are not in P and are not NP-complete, they seem to lack any interesting structure which would allow us to give any justification for these conjectures. Instead, in this paper, we attack these two problems in the form of relativization. The relativization of *MINKT* and *MINLT* is straightforward: the minimum Turing machines we search for are allowed to make queries to oracles. In this relativized form, we are able to show that some conjectured properties of *MINKT* do hold relative to some oracles and hence to give some support to the conjectures.

To state our main results, we need to define some subproblem of *MINKT*:

$$KT^B[s(n), t(n)] = \{x : \langle x, 0^{s(|x|)}, 0^{t(|x|)} \rangle \in MINKT^B\}.$$

Our main results can be summarized as follows:

(1) Let $0 < a < b < 1$. There exists an oracle A such that the problems $KT^A[an, t(n)]$ and $\overline{KT^A[bn, t(n)]}$ are polynomial-time inseparable (and so $MINKT^A$

is not in P^A), and there exists an oracle B such that the problems $KT^B[an, t(n)]$ and $\overline{KT^B[bn, t(n)]}$ are polynomial-time separable.

(2) There exists an oracle A such that $MINKT^A$ is not in $co\text{-}NP^A$ and is not $\leq_T^{P,A}$-complete for NP^A, where $\leq_T^{P,A}$ is the polynomial-time Turing reduction relative to A; on the other hand, there exists an oracle B such that $MINKT^B$ is $\leq_T^{SNP,B}$-complete for NP^B, while $NP^B \neq co\text{-}NP^B$, where $\leq_T^{SNP,B}$ is the relativized strong NP reduction [19].

The first result shows that the problem $MINKT$, in the relativized form, could be either easy or hard to solve and so implies that the unrelativized problem $MINKT$ is probably not polynomial-time solvable, and its solution must use an unrelativizable proof technique. The second result provides some evidence for the conjecture that $MINKT$ is a problem in NP that is neither in P nor NP-complete. It also suggests an interesting possibility: that $MINKT$ is not polynomial-time Turing complete for NP but is complete under some weaker reducibilities. We remark that there have been some natural problems proven to be complete for NP under some weak reducibilities (e.g., Adleman and Manders [1], Vazirani and Vazirani [26], and Chung and Ravikumar [6]). However, no evidence has been given to show that they are not Turing-complete for NP. Our result is, to the best of our knowledge, the only problem which admits a natural relativization and hence can be shown to be not $\leq_T^P$-complete in the relativized form.

In addition to these results, we also investigate other properties of the problems $MINKT^A$ and $KT^A[an, t(n)]$. Call a set X *$\mathcal{C}$-immune* if X is infinite and X has no infinite subset in $\mathcal{C}$. A set X being P-immune implies that any polynomial heuristic algorithm for X can only solve finitely many instances in X. We study the question of whether $\overline{KT[an, t(n)]}$ is P-immune, where $0 < a < 1$ and t is a polynomial function. Intuitively, it is difficult to generate strings that have high program-size complexity, and it is provable that $\overline{KT[an, n^{\log n}]}$ is $\mathcal{C}$-immune, where $\mathcal{C}$ is the class of P-printable sets (a set X is *P-printable* if the set $X \cap \{0,1\}^n$ is printable from 0^n in polynomial time). We generalize this idea to show that there exists an oracle A such that $\overline{KT^A[an, t(n)]}$ is NP^A-immune. On the other hand, there exists an oracle B such that $\overline{KT^B[an, t(n)]}$ is not in P^B and is not P^B-immune. Therefore, the question of whether $\overline{KT^A[an, t(n)]}$ is P-immune can be relativized in both ways and needs unrelativizable proof techniques to solve it in the unrelativized form.

Proofs of the above results all involve two basic principles about KT-complexity of a string relative to a set:

(1) If a string x is *random* relative to A, i.e., its KT-complexity is high, then a deterministic polynomial-time oracle Turing machine cannot query about x on input y of smaller KT-complexity (cf. Hartmanis [8]).

(2) When we construct an oracle A, it is easy to make some specific strings x have low KT-complexity by encoding x into set A.

The above simple observations allow us to construct oracles A to satisfy requirements about KT-complexity of some specific strings. However, the construction often becomes more difficult when the requirements are too strong. The following observations seem to indicate the limit of this type of construction:

(3) A string x having low KT-complexity relative to set A is difficult to convert to a string with high KT-complexity if we are allowed to make only local changes on set A.

(4) A string x having a high KT-complexity relative to A may be inadvertently converted to a string with low KT-complexity if we change the membership of strings

of lower KT-complexity.

(5) Most strings have high KT-complexity relative to any set A. Therefore, the principle (2) above can be applied to only a sparse set of strings.

All of the results stated above also hold with respect to problem *MINLT*. Indeed, there seems to be a simple transformation of the proofs of the results about *MINKT* to the proofs of analogous results about *MINLT*. This observation supports our viewpoint of treating the problem *MINKT* as a simpler version of *MINLT*, and suggests an interesting link between program-size complexity and learning in the polynomial-time setting.

2. Preliminaries. We will deal with finite strings over $\{0,1\}$. We let λ denote the empty string. We let $|x|$ denote the length of a string x. For any set A, we let χ_A denote its characteristic function; i.e., $\chi_A(x) = 1$ if $x \in A$ and $\chi_A(x) = 0$ if $x \notin A$. We write $A^{\leq n}$ to denote the set $A \cap \{x \in \{0,1\}^* : |x| \leq n\}$.

We use the standard model of multitape oracle Turing machines (TMs). In particular, we assume that the query tape is cleaned after each query. We assume that the reader is familiar with the basic notions in complexity theory. In particular, we let P^A (NP^A) be the class of languages over $\{0,1\}$ which are computable by some oracle TMs (nondeterministic oracle TMs, respectively) in polynomial time relative to set A. For each oracle A and each machine M, we write $L(M, A)$ to denote the set of inputs accepted by machine M using oracle A. A set B is $\leq_T^{P,A}$-complete for NP^A if $B \in NP^A$ and for all $C \in NP^A$, $C = L(M, A \oplus B)$ for some polynomial-time oracle TM M, where $A \oplus B$ denotes the set $\{0x : x \in A\} \cup \{1y : y \in B\}$. A weaker notion of completeness is also used: A set B is $\leq_T^{SNP,A}$-complete for NP^A if $B \in NP^A$ and for all $C \in NP^A$, $\overline{C} \in L(N, A \oplus B)$, for some polynomial-time nondeterministic oracle TM N [19].

Let T be an oracle TM. The *program-size complexity* of (printing) a string x relative to set A is defined as

$$K_T^A(x) = \min\{|\alpha| : T^A(\alpha) \text{ prints } x\},$$

where $\min(\emptyset)$ is defined to be ∞. We say α is a minimum *program* computing x. The *time-bounded program-size complexity* of x relative to A is the length of a minimum program which computes x within a specific number of moves. More precisely,

$$KT_T^A(x,t) = \min\{|\alpha| : T^A(\alpha) \text{ prints } x \text{ in } t \text{ moves}\}.$$

We call this value the KT-complexity of x with time bound t relative to set A. The importance of the notion of KT-complexity has been recognized as early as in Kolmogorov's original 1965 paper on program-size complexity [15].

It is well known that the KT-complexity is invariant when we replace the machine T by a universal oracle TM.

PROPOSITION 2.1. *There exist a universal oracle* TM U *and a polynomial* p_0 *such that for any oracle set* A *and any input* α, *if* $T^A(\alpha)$ *halts and prints* x *in* t *moves, then* $U^A(\langle \tilde{T}, \alpha \rangle)$ *halts and prints* x *in* $p_0(t)$ *moves, where* $\tilde{T}$ *encodes the oracle* TM T. (*The polynomial* p_0, *of course, depends on the specific model of* TMs *that we use. In general, it seems safe to assume that* $p_0(n) = O(n \log n)$.)

In order to calculate the input length to machine U, we need to specify how the two inputs $\tilde{T}$ and α are encoded into a single string. For any string x, let $d(x)$ denote the string which doubles each bit of x; i.e., $d(0) = 00$, $d(1) = 11$, and $d(ax) = d(a)d(x)$ for all $a \in \{0,1\}$ and $x \in \{0,1\}^*$. We define $\langle x, y \rangle$ to be $d(x)10y$. Thus,

$|\langle x,y\rangle| = 2|x| + |y| + 2$. For more than two strings, we define $\langle x_1, x_2, \cdots, x_m\rangle$ to be $d(x_1)01d(x_2)01\cdots 01d(x_{m-1})10x_m$. Thus, $|\langle x_1, x_2, \cdots, x_m\rangle| = \sum_{i=1}^{m-1}(2|x_i|+2)+|x_m|$. Using this coding scheme, we obtain a simple relation between KT-complexities with respect to different TMs.

PROPOSITION 2.2. *Let the universal* TM U *and polynomial* p_0 *be defined as in Proposition* 2.1. *Then, for all sets* A *and all strings* x,

$$KT_U^A(x, p_0(t)) \leq KT_T^A(x,t) + c_T,$$

where c_T *is a constant depending on the machine* T.

We will, in the rest of the paper, write $KT^A(x,t)$ to denote $KT_U^A(x,t)$, and will let p_0 be the corresponding polynomial function. We also write $K^A(x)$ to denote $\min_{t>0} KT^A(x,t)$.

Let A be a set and c an integer. A string x is said to be *c-random* relative to A if $K^A(x) \geq |x| - c$. We observe that for any set A and any integer n, *most* strings of length n are c-random relative to A if $c \geq 1$.

PROPOSITION 2.3. *For any set* A *and any* n, *there are at least* $2^n - 2^{n-c}$ *many strings of length* n *that are* c*-random relative to* A.

The fundamental question about KT-complexity of a given string x can be formulated into the following set:

$$MINKT^A = \{\langle x, 0^s, 0^t\rangle : KT^A(x,t) \leq s\}.$$

It is clear that $MINKT^A$ is in the class NP^A. It is not known whether $MINKT^\emptyset$ is NP-complete. The following in a class of subproblems of $MINKT^A$:

$$KT^A[s(n), t(n)] = \{x : \langle x, 0^{s(|x|)}, 0^{t(|x|)}\rangle \in MINKT^A\}.$$

If $0 < s(n) < n$ and $t(n)$ is a polynomial function, then $KT^A[s(n), t(n)]$ is in NP^A and is not known to be in P^A or to be NP^A-complete.

Next we consider the program-size complexity of (recognizing) two lists of strings. We write $\vec{y}$ to denote a finite list of strings $\vec{y} = \langle y_1, \cdots, y_m\rangle$. For convenience, we also let $\vec{y}$ denote the set of strings in the list $\vec{y}$; i.e., $\vec{y} = \{y_1, \cdots, y_m\}$. We modify our model of oracle TMs into recognizers so that each machine can take two inputs α and β and they are put in two input tapes (rather than encoded into a single string and put in one input tape). Let R be a two-input oracle TM. We define $LT_R^A(\langle\vec{y}, \vec{z}\rangle, t)$ to be the length of the minimum string α such that for all y in $\vec{y}$, $R^A(\alpha, y)$ halts and accepts in t moves, and for all z in $\vec{z}$, $R^A(\alpha, z)$ halts and rejects in t moves. We call this complexity the LT-complexity of the lists $\langle\vec{y}, \vec{z}\rangle$. Note that for the sake of simplicity, we made the runtime a uniform value t for all y_i's in $\vec{y}$ and all z_i's in $\vec{z}$. A more general way to define the LT-complexity is to use a different t_i for each y_i or z_i. We remark that our results, to be proved in the following sections, remain true using this general definition.

Similar to the KT-complexity, LT-complexity is invariant under a universal oracle TM.

PROPOSITION 2.4. *There exist a universal two-input oracle* TM V *and a polynomial* p_1 *such that for any oracle set* A *and any input* (α, w), *if* $R^A(\alpha, w)$ *halts and accepts* (*or rejects*) *in* t *moves, then* $V^A(\langle\tilde{R}, \alpha\rangle, w)$ *halts and accepts* (*or rejects, respectively*) *in* $p_1(t)$ *moves.* (*Again, we may assume that* $p_1(n) = O(n \log n)$.)

PROPOSITION 2.5. *Let the universal* TM V *and polynomial* p_1 *be defined as in Proposition* 2.4. *Then, for all sets* A *and all lists* $\langle\vec{y}, \vec{z}\rangle$,

$$LT_V^A(\langle\vec{y}, \vec{z}\rangle, p_1(t)) \leq LT_R^A(\langle\vec{y}, \vec{z}\rangle, t) + d_R,$$

where d_R *is a constant depending on the machine* R.

We fix a universal TM V and its corresponding p_1 and write $LT^A(\langle\vec{y},\vec{z}\rangle, t)$ to denote $LT_V^A(\langle\vec{y},\vec{z}\rangle, t)$.

The problem about LT-complexity that we are interested in is:

$$MINLT^A = \{\langle\langle\vec{y},\vec{z}\rangle, 0^s, 0^t\rangle : LT^A(\langle\vec{y},\vec{z}\rangle, t) \leq s\}.$$

It is clear that $MINLT^A$ is in NP^A. Let EL_m be the set of all pairs $\langle\vec{y},\vec{z}\rangle$ such that $\vec{y}$ and $\vec{z}$ contain only strings of length m. Let $EL = \cup_{m=1}^{\infty} EL_m$. A class of subproblems of interest is the following:

$$LT^A[s(n), t(n)] = \{\langle\vec{y},\vec{z}\rangle \in EL_m : \langle\langle\vec{y},\vec{z}\rangle, 0^{s(m)}, 0^{t(m)}\rangle \in MINLT^A, m \geq 1\}.$$

In the above definition, for the sake of convenience, we consider only pairs $\langle\vec{y},\vec{z}\rangle$ in which all strings are of the same length. It should be pointed out that this restriction is not too strong because arbitrary pairs $\langle\vec{y},\vec{z}\rangle$ consisting of strings of different lengths can be encoded into pairs $\langle\vec{y},\vec{z}\rangle$ in EL by a simple padding scheme. Note that $LT^A[s(n), t(n)] \in NP^A$ if both $s(n)$ and $t(n)$ are polynomial functions.

The proofs of our main results make use of some specific Turing machines. We define them in the following.

DEFINITION 2.6. For any set A, $\xi_A(n, w) = \chi_A(w01)\chi_A(w01^2)\cdots\chi_A(w01^n)$. Let T_0 be the oracle TM that, on input $\langle n, k, w\rangle$, with oracle A, queries the oracle to print $\xi_A(n, w0^k)$. For convenience, also let T_1 be the oracle TM such that $T_1^A(\langle n, w\rangle) = T_0^A(\langle n, 0, w\rangle)$.

We assume that there is a polynomial t_1 such that $T_0^A(\langle n, k, w\rangle)$ halts in $t_1(n+k+|w|)$ moves. (It seems reasonable to assume that $t_1(n) = O(n^2)$.) We fix a polynomial $t_0(n) = p_0(t_1(4n))$ so that U^A can simulate T_0 in time $t_0(n)$. We also assume that $t_0(n)$ is big enough so that U^A can do, in time $t_0(n)$, some trivial tasks such as computing the identity function and converting the binary representation of an integer k to 0^k.

DEFINITION 2.7. Let R_1 be the two-input oracle TM that, on input $(\langle m, w\rangle, z)$ and with oracle A, accepts if and only if $w0^m z \in A$.

We assume that there is a polynomial r_1 such that $R_1^A(\langle m, w\rangle, z)$ halts in $r_1(m + |w| + |z|)$ moves. (Again, it seems reasonable to assume that $r_1(n) = O(n^2)$.) We let $r_0(n) = p_1(r_1(4n))$. We also assume that V^A can simulate, in time $r_0(n)$, some trivial machines, such as a machine that encodes and accepts a fixed list of strings and rejects all other strings.

3. Polynomial-time computability. Intuitively, it does not appear that the problems *MINKT* and *MINLT* can be solved in polynomial time. In this section we are going to show that a very general subproblem of $MINKT^A$ is indeed not solvable in polynomial time relative to some A. On the other hand, we also show that this subproblem of $MINKT^B$ is solvable in polynomial time relative to some other set B. The same results hold for the problem *MINLT*, in a slightly weaker form.

The subproblem here is to distinguish between the class $KT[an, t(n)]$ and the class $\overline{KT[bn, t(n)]}$, where $0 < a < b < 1$ and $t(n)$ is a polynomial greater than the polynomial $t_0(n)$ of Definition 2.6. We say that two sets B and C are *P^A-separable* if $B \cap C = \emptyset$ and there exists a set $X \in P^A$ such that $B \subseteq X$ and $C \subseteq \overline{X}$. Sets B and C are *P^A-inseparable* if $B \cap C = \emptyset$, but they are not P^A-separable. We first show that, relative to some oracle A, $KT^A[an, t(n)]$ and $\overline{KT^A[bn, t(n)]}$ are P^A-inseparable if $t(n) \geq t_0(n)$, where t_0 is defined in Definition 2.6.

THEOREM 3.1. *Let* $0 < a < b < 1$ *and* t *be a polynomial such that* $t(n) \geq t_0(n)$. *There exists a set* A *such that* $KT^A[an, t(n)]$ *and* $\overline{KT^A[bn, t(n)]}$ *are* P^A*-inseparable.*

Proof. Assume that $\{M_i\}$ is an enumeration of polynomial-time oracle TMs. For each i, let q_i be a polynomial bounding the runtime of machine M_i. We need to show that for each $i \geq 0$, the following requirement is satisfied:

$$R_i : (\exists x_i)[M_i^A(x_i) = 1 \text{ and } x_i \notin KT^A[bn, t(n)]] \\ \text{or } \quad [M_i^A(x_i) = 0 \text{ and } x_i \in KT^A[an, t(n)]].$$

We construct set A by stages such that requirement R_i is satisfied in the ith stage. We let $A_{-1} = \emptyset$ and $e_{-1} = 0$.

Stage i. Assume that e_{i-1} and A_{i-1} have been defined earlier such that all strings in A_{i-1} are of length less than or equal to e_{i-1}. We choose an integer $n = n_i$ that satisfies $an > 2e_{i-1}$, $(1-b)n > 1$, and $an > 2\log(q_i(n)) + 4\log n + 4 + 2c_{T_1}$, where c_{T_1} is the constant associated with machine T_1 of Definition 2.6. Let $x = x_i$ be the least string of length n that is 1-random relative to A_{i-1}. Simulate machine M_i on input x with oracle A_{i-1}. If it accepts, then we let $A_i = A_{i-1}$ and $e_i = \max\{q_i(n), t(n)\}$, and go to the next stage.

If $M_i^{A_{i-1}}(x)$ rejects, then we find a string w of length $an/2$ such that no string in set $w\{0,1\}^*$ has been queried in the computation of $M_i^{A_{i-1}}(x)$. (Note that we chose n such that $q_i(n) < 2^{an/2}$ and so such a w must exist.) Then, we let

$$A_i = A_{i-1} \cup \{w01^j : \text{ the } j\text{th bit of } x \text{ is } 1\},$$

and $e_i = \max\{q_i(n), t(n)\}$, and go to the next stage.

End of Stage i.

We let $A = \cup_{i=0}^{\infty} A_i$. We claim that A satisfies all requirements R_i. First, assume that $M_i^{A_{i-1}}(x_i)$ accepts in Stage i. Note that we always choose n_j such that $an_j/2 > e_{j-1} \geq e_i$, if $j > i$, and so we never add any string of length less than or equal to e_i to A in later stages. This implies that $M_i^A(x_i)$ behaves the same as $M_i^{A_{i-1}}(x_i)$ and it accepts. In addition, $e_i \geq t(n_i)$ implies that for any string α, $U^A(\alpha)$ behaves the same as $U^{A_{i-1}}(\alpha)$ in $t(n_i)$ moves. Thus the fact that x_i is 1-random relative to A_{i-1} implies that for no α of length less than or equal to $bn < n-1$, $U^{A_{i-1}}(\alpha)$ prints x_i in $t(n_i)$ moves and hence $x_i \notin KT^A[bn, t(n)]$.

Next assume that $M_i^{A_{i-1}}(x_i)$ rejects in Stage i. Then, $A_i - A_{i-1} \subseteq w\{0,1\}^*$, and the computation of $M_i^{A_{i-1}}(x_i)$ never queries any string in this set. It follows that $M_i^A(x_i)$ also rejects. On the other hand, we observe that $\xi_A(n_i, w) = x_i$ and so machine T_1, with oracle A, prints x_i on input $\langle n_i, w\rangle$ in $t_1(n_i)$ moves. The length of $\langle n_i, w\rangle$ is bounded by $2\log n_i + an_i/2 + 2 \leq an_i - c_{T_1}$. Thus, $KT^A(x, t(n_i)) \leq KT_{T_1}^A(x, t_1(n_i)) + c_{T_1} \leq an_i$, and so $x_i \in KT^A[an, t(n)]$. This completes the proof. □

Next we show that, relative to some oracle A, $KT^A[an, t(n)]$ and $\overline{KT^A[bn, t(n)]}$ are P^A-separable for a sufficiently large polynomial t. Note that if $P^A = NP^A$, then both $KT^A[an, t(n)]$ and $KT^A[bn, t(n)]$ are in P^A and so the P^A-separability result would be trivial. In the following, we show that even if $P^A \neq NP^A$, it is still possible to separate $KT^A[an, t(n)]$ from $\overline{KT^A[bn, t(n)]}$ in polynomial time relative to A.

The following specific oracle TMs are useful in the next proof, as well as later proofs.

(1) Let T_2 be the oracle TM that, on input $\langle m, \alpha\rangle$ and with oracle A, works as follows: It simulates $U^A(\alpha)$ until the mth move; if the mth move of $U^A(\alpha)$ makes a query "$y \in ?A$," then it outputs y else it outputs 0.

(2) Let T_3 be the oracle TM that, on input $\langle i, j, m, \alpha\rangle$ and with oracle A, first obtains $x = T_2^A(\langle m, \alpha\rangle)$, and then prints the substring y of x from the ith bit to the jth bit.

Let c_{T_2} and c_{T_3} be the constants associated with these machines as defined in Proposition 2.2.

THEOREM 3.2. *Let $0 < a < b < 1$ and t be a polynomial such that $t(n) \geq t_0(n)$. There exists a set A such that $P^A \neq NP^A$ and $KT^A[an, t(n)]$ and $\overline{KT^A[bn, t(n)]}$ are P^A-separable.*

Proof. To separate the classes $KT^A[an, t(n)]$ from $\overline{KT^A[bn, t(n)]}$, we encode the information on whether $x \in KT^A[an, t(n)]$ in $\chi_A(f(x))$, where $f(x) = 0^m x$, $m = 2t(|x|) - |x| + 1$. Thus our requirements are:

$$R_{0,x} : [x \in KT^A[an, t(n)] \Rightarrow f(x) \in A] \quad \text{and} \quad [x \notin KT^A[bn, t(n)] \Rightarrow f(x) \notin A].$$

To satisfy the requirement $P^A \neq NP^A$, we define $L_A = \{0^n : (\exists x, |x| = n)\ x \in A\}$ and satisfy the following subrequirements:

$$R_{1,i} : (\exists n)[0^n \in L(M_i, A) \iff 0^n \notin L_A],$$

where $\{M_i\}$ is an enumeration of polynomial-time oracle TMs.

We satisfy all these requirements by stages. First, let $A_{-1} = A'_{-1} = \emptyset$ and $e_{-1} = 0$. Let $c = (a+b)/2$. In Stage i, we will define integer e_i and satisfy requirements $R_{0,x}$ for all x such that $e_{i-1} < |x| \leq e_i$, and satisfy requirement $R_{1,i}$ using a witness 0^n, $e_{i-1} < n \leq e_i$. Assume that the machine M_i has a polynomial time bound q_i.

Stage i. Assume that A_{i-1}, A'_{i-1} and e_{i-1} have been defined in earlier stages such that $A_{i-1} \cap A'_{i-1} = \emptyset$. Choose an even integer $n > e_{i-1}$ such that $an > 2\log(q_i(n))$ and $(b-a)n/4 \geq 6\log(t(n)) + \log(q_i(n)) + c_{T_0} + c_{T_2} + c_{T_3} + 7$. Let $B = A_{i-1}$ and $B' = A'_{i-1}$.

(1) First, for all x, $|x| \leq n-1$ and $e_{i-1} < |f(x)|$, do the following in increasing order: Simulate to see whether $x \in KT^B[an, t(n)]$. If yes, then let $B = B \cup \{f(x)\}$; otherwise, let $B' = B' \cup \{f(x)\}$.

(2) Simulate $M_i^A(0^n)$ with the queries "$y \in ?A$" answered as follows:

(2.a) If $y \in B$, then answer YES; if $y \in B'$, then answer NO.

(2.b) If $y \notin B \cup B'$ and $y \neq f(x)$ for any x, then answer NO and let $B' = B' \cup \{y\}$.

(2.c) If $y \notin B \cup B'$ and $y = f(x)$ for some x, then answer YES and let $B = B \cup \{y\}$.

When the simulation is done, let $B_0 = B$ and $B'_0 = B'$.

(3) If the simulation of $M_i^A(0^n)$ in step (2) accepts, then do nothing. Otherwise, find a string x_0 of length n which is 1-random relative to B_0 and which is not in B'_0. (Note that there are at least 2^{n-1} many 1-random strings relative to B_0 and that $B'_0 \cap \{x : |x| = n\}$ has size at most $q_i(n)$. Since we have chosen n such that $2^{n-1} > 2^{an} > q_i(n)$, such an x_0 must exist.) Let $B = B_0 \cup \{x_0\}$.

(4) For each x, $|x| \geq n$, if $y = f(x)$ has been queried in step (2), case (2.c), then do the following: Find a string w_x of length $\lfloor cn \rfloor$ such that w_x is 1-random relative to B_0, $x_0 \notin 1w_x\{0,1\}^*$, $w_x \neq w_z$ for all w_z defined before in this step, and that none of the strings in $1w_x\{0,1\}^*$ is queried in step (2). (Again, we have $2^{an} > (q_i(n))^2$, and this implies the existence of w_x's.) Let $B = B \cup \{1w_x 0^{n+1}1^j : \text{the } j\text{th bit of } x \text{ is } 1\}$ and $B' = B' \cup \{1w_x 0^{n+1}1^j : \text{the } j\text{th bit of } x \text{ is } 0\}$.

(5) For all other x, $n \le |x| \le \max\{q_i(n), 2t(n)\}$, but $f(x)$ was not queried in step (2), we simulate to see whether $x \in KT^B[an, t(n)]$ and let $B = B \cup \{f(x)\}$ if yes, and $B' = B' \cup \{f(x)\}$ otherwise. (Note that strings in $1w_x\{0,1\}^*$ begin with 1 and so cannot contain $f(x)$.) Let $A_i = B$, $A_i' = B'$, and $e_i = \max\{m : (\exists y, |y| = m)\ y \in B \cup B'\}$.

End of Stage i.

Let $A = \cup_{i=0}^{\infty} A_i$. We claim that A satisfies all requirements. We first verify that in Stage i, requirement $R_{1,i}$ is satisfied. Note that in step (2) of the simulation for $M_i^A(0^n)$, we put all answers to the queries into either B or B' and then never change their memberships later. So the simulation result must be correct. Also, if $M_i^A(0^n)$ accepts, then $A \cap \{x : |x| = n\} = \emptyset$ (note that n is even but $|f(x)|$ is odd for all x, and that all strings of the form $1w_x0^{n+1}1^j$ added to B' in step (4) are longer than n), and if $M_i^A(0^n)$ rejects, then we have an x_0 of length n in A. This shows that requirement $R_{1,i}$ is satisfied.

Next we show that if $e_{i-1} < |x| < n$ (and so the membership of $f(x)$ in A is determined in step (1) of Stage i) then requirement $R_{0,x}$ is satisfied. First, observe that after Stage i, we only add strings of length $\ge e_i \ge |f(x)| \ge t(|x|)$ to A, and so later stages will not affect the membership of x in $KT^A[an, t(n)]$. Then we see that later in step (1) (after determining the membership of $f(x)$ in A) and in step (2), we only add strings longer than $t(|x|)$ to A and so they do not change the membership of x in $KT^A[an, t(n)]$.

In steps (3) and (4), we may add some strings of length less than $t(|x|)$ to A. Namely, in step (3), we may add to A a string $z = x_0$ of length n that is 1-random relative to B_0. In step (4), we may add strings to A of the form $z = 1w0^{n+1}1^j$, $j \le q_i(n)$, where w is of length $\lfloor cn \rfloor$ and is 1-random relative to B_0.

We claim that for all strings α of length less than or equal to an, $U^{B_0}(\alpha)$ cannot query such a string z ($z = x_0$ or $z = 1w0^{n+1}1^j$) in $t(n)$ moves. Suppose otherwise that $U^{B_0}(\alpha)$ first queries such a string z at the mth move, $m \le t(n)$. First assume that $z = x_0$. Consider the machine T_2 defined above. We see that $T_2^{B_0}(\langle m, \alpha \rangle)$ prints x_0, since in the first $m-1$ moves, $U^{B_0}(\alpha)$ does not query about these special strings. So,

$$K_{T_2}^{B_0}(x_0) \le 2|m| + |\alpha| + 2 \le an + 2\log(t(n)) + 2.$$

This implies that

$$K^{B_0}(x_0) \le an + 2\log(t(n)) + 2 + c_{T_2} < n - 1.$$

This contradicts the fact that x_0 is 1-random relative to B_0. Similarly, for $z = 1w0^{n+1}1^j$, $j \le q_i(n)$, machine T_3 prints w from input $\langle 2, |w| + 1, m, \alpha \rangle$, using oracle B_0, if $U^{B_0}(\alpha)$ queries about $1w0^{n+1}1^j$ at the mth move. Again, a contradiction can be derived. So the claim is proven.

The above shows that the simulation of whether $x \in KT^B[an, t(n)]$ in step (1) is correct relative to oracle A. So, requirement $R_{0,x}$ is satisfied for x such that $|x| < n$.

For x of length between n and $\max\{q_i(n), 2t(n)\}$, we determine the membership of $f(x)$ in either step (2), case (2.c), or step (5). If the membership of $f(x)$ in A is determined in step (5) then, similar to the above argument, all strings added to A later are of length greater than $t(|x|)$ and so would not affect the requirement $R_{0,x}$. If the membership of $f(x)$ in A is determined in step (2), then we added in step (4) strings $1w_x0^{n+1}1^j$ to A if the jth bit of x is 1. Thus, x can be printed from the input $\langle |x|, n, 1w_x \rangle$ by the machine T_0 of Definition 2.6 in $t_1(3|x|)$ moves. So,

$KT^A_{T_0}(x, t_1(3|x|)) \le 2\log(|x|) + 2|n| + cn + 6 \le b|x| - c_{T_0}$, and so $KT^A(x, t(|x|)) \le b|x|$. Thus requirement $R_{0,x}$ is satisfied. This completes the proof. □

Remarks. (1) The above proof also works for P^A-separability of, for instance, $KT^A[n/2, t_0(n)]$ from $\overline{KT^A[n/2 + \log^2 n, t_0(n)]}$, with $P^A \neq NP^A$. Whether it can be improved to show that $KT^A[n/2, t_0(n)] \in P^A \neq NP^A$ is not known. Our construction above fails because we need, in step (4), to encode string $y = f(x)$ into set A such that the information can be decoded using a program of length less than or equal to bn and yet cannot be decoded by any program of length less than or equal to an (so that the requirements established in step (1) are not violated). This presents two inconsistent requirements if $a = b$.

(2) The construction above for set A is not recursive, because we need to determine whether a string x_0 is 1-random relative to B_0 in step (3). It can be made recursive by replacing the absolute 1-randomness in the construction by 1-randomness with respect to a time bound t, where t is a large value, because all the arguments above involve only time-bounded simulations between machines T_1, T_2, T_3, and U.

Next we consider the polynomial-time computability of $MINLT^A$. Our results on $MINLT^A$ are similar to Theorems 3.1 and 3.2. Namely, we show that if $0 < a < b < 1$, then, depending on oracles A, $LT^A[an, t(n)]$ and $\overline{LT^A[bn, r(n)]}$ could be P^A-separable or P^A-inseparable, if $r(n)$ is sufficiently larger than $t(n)$. The P^A-separability result is a little weaker than Theorem 3.2, because we require that $r(n)$ is larger than $t(n)$.

Recall that r_1 is the polynomial runtime of machine R_1 of Definition 2.7 and that $r_0(n) = p_1(r_1(n))$.

THEOREM 3.3. *Let $0 < a < b < 1$ and t be a polynomial such that $t(n) \ge r_0(n)$. There exists a set A such that $LT^A[an, t(n)]$ and $\overline{LT^A[bn, t(n)]}$ are P^A-inseparable.*

Proof. The proof is a slight modification of the proof of Theorem 3.1. We only give a sketch. In Stage i, we try to satisfy the requirement

$$R_i : (\exists \langle \vec{y}, \vec{z} \rangle)[M_i^A(\langle \vec{y}, \vec{z} \rangle) = 1 \text{ and } \langle \vec{y}, \vec{z} \rangle \notin LT^A[bn, t(n)]] \\ \text{or } \quad [M_i^A(\langle \vec{y}, \vec{z} \rangle) = 0 \text{ and } \langle \vec{y}, \vec{z} \rangle \in LT^A[an, t(n)]],$$

where M_i is the ith polynomial-time oracle TM. Let q_i bound the runtime of M_i.

Assume the same setting in Stage i as in Theorem 3.1. Choose a sufficiently large integer $n > e_{i-1}$. Let x be a string of length n that is 1-random relative to A_{i-1}. Define, for each $j \le 2^n$, σ_j^n to be the n-bit binary representation of integer j. Define $\vec{y} = \{\sigma_j^n : \text{the } j\text{th bit of } x \text{ is } 1\}$, and $\vec{z} = \{\sigma_j^n : \text{the } j\text{th bit of } x \text{ is } 0\}$. Note that $\langle \vec{y}, \vec{z} \rangle \in EL_n$.

Now, simulate $M_i^{A_{i-1}}(\langle \vec{y}, \vec{z} \rangle)$. If it accepts, then do nothing. Otherwise, we find a string w of length $an/2$ such that no string in set $w\{0,1\}^*$ has been queried in the computation of $M_i^{A_{i-1}}(\langle \vec{y}, \vec{z} \rangle)$. Then, let

$$A_i = A_{i-1} \cup \{w\sigma_j^n : \text{ the } j\text{th bit of } x \text{ is } 1\}.$$

This ends the construction of Stage i.

We first observe that the simulation outcome of $M_i^{A_{i-1}}(\langle \vec{y}, \vec{z} \rangle)$ is the same as $M_i^A(\langle \vec{y}, \vec{z} \rangle)$, because all strings in $A_i - A_{i-1}$ are not queried in the computation of $M_i^A(\langle \vec{y}, \vec{z} \rangle)$. Next, we claim that if $M_i^{A_{i-1}}(\langle \vec{y}, \vec{z} \rangle)$ accepts, then $\langle \vec{y}, \vec{z} \rangle \notin LT^A[bn, t(n)]$. Assume otherwise that there exists an α of length less than or equal to bn such that $V^A(\alpha, y)$ accepts in $t(n)$ moves for all $y \in \vec{y}$ and $V^A(\alpha, z)$ rejects in $t(n)$ moves for all $z \in \vec{z}$. It is easy to see that by choosing a large e_i we can make sure that $A - A_{i-1}$ contains no string of length less than or equal to $t(n)$. So, the above also holds for

oracle A_{i-1}. Consider machine T_4 defined as follows: $T_4^A(\langle m, \alpha\rangle)$ simulates $V^A(\alpha, \sigma_j^m)$ for all $j = 1, \cdots, m$, and prints the outcomes in the increasing order.

We can see that machine T_4, with oracle A_{i-1}, prints x on input $\langle n, \alpha\rangle$. Thus, $K^{A_{i-1}}(x) \leq bn + 2\log n + 2 + c_{T_4} \leq n - 1$ when n is chosen large enough. This contradicts the 1-randomness of x relative to A_{i-1}.

Finally, we observe that if the simulation of $M_i^{A_{i-1}}(\langle \vec{y}, \vec{z}\rangle)$ rejects, then $M_i^A(\langle \vec{y}, \vec{z}\rangle)$ rejects, and furthermore we claim that $\langle \vec{y}, \vec{z}\rangle \in LT^A[an, t(n)]$. To see this, we note that the machine R_1^A of Definition 2.7 accepts input $(\langle 0, w\rangle, \sigma_j^n)$ in $r_1(2n)$ moves if and only if $w\sigma_j^n \in A$ if and only if the jth bit of x is 1. This implies that $LT_{R_1}^A(\langle \vec{y}, \vec{z}\rangle, r_1(2n)) \leq an/2 + 2\log n + 2$ and so $LT^A(\langle \vec{y}, \vec{z}\rangle, r(n)) \leq an$ if $an/2 > d_{R_1} + 2\log n + 2$. □

THEOREM 3.4. *Let $0 < a < b < 1$ and let $t(n), r(n)$ be two polynomials such that $r(n) \geq r_0(t(n) + 1)$. There exists a set A such that $P^A \neq NP^A$ and $LT^A[an, t(n)]$ and $\overline{LT^A[bn, r(n)]}$ are P^A-separable.*

Proof. Again, the proof is similar to that of Theorem 3.2 and we only give a sketch. Recall that EL_m contains $\langle \vec{y}, \vec{z}\rangle$ in which all strings are of length m, and that $EL = \cup_{m=1}^{\infty} EL_m$. For each $x \in EL$, we encode the information about $x \in LT^A[an, t(n)]$ into $f(x) = o^\ell x$, $\ell = 4t(|x|) - |x| + 1$.

At Stage i, we will satisfy the requirement

$$R_{1,i} : (\exists n)[M_i^A(0^n) = 1 \iff 0^n \notin L_A],$$

where M_i is the ith polynomial-time oracle TM and $L_A = \{0^n : (\exists y, |y| = 3n)\ y \in A\}$. (Note that we have changed the definition of L_A so that the witness for $0^n \in L_A$ must have length $3n$.) In the meantime, for all x, $e_{i-1} < |x| \leq e_i$, we satisfy

$$R_{0,x} : [x \in LT^A[an, t(n)] \Rightarrow f(x) \in A] \quad \text{and} \quad [x \notin LT^A[bn, r(n)] \Rightarrow f(x) \notin A].$$

Stage i. Assume that e_{i-1}, A_{i-1}, and A'_{i-1} have been defined as in Theorem 3.2. We let $c = (a + b)/2$, $B = A_{i-1}$, and $B' = A'_{i-1}$. We choose a sufficiently large even integer n and perform the following:

(1) For each x, if $x \in EL_m$, $m < n$, then simulate to see if $x \in LT^B[an, t(n)]$. Add $f(x)$ to B if yes, and add $f(x)$ to B' otherwise.

(2) Simulate $M_i^A(0^n)$ and answer the query "$u \in ?A$" as follows:

(2.a) If $u \in B$, then answer YES; if $u \in B'$, then answer NO.

(2.b) If $u \notin B \cup B'$ and $u \neq f(x)$ for any $x \in EL$, then answer NO and add u to B'.

(2.c) If $u \notin B \cup B'$ and $u = f(x)$ for some $x \in EL$, then answer YES and add u to B.

When the simulation is done, let $B_0 = B$ and $B'_0 = B'$.

(3) If the simulation of $M_i^A(0^n)$ above accepts, then do nothing. Otherwise, find x_0 of length $3n$ that is not in B' and is 1-random relative to B_0. Add x_0 to B.

(4) For each $x \in EL_m$, $m \geq n$, if $u = f(x)$ is queried and answered YES in step (2), then find a w_x of length $\lfloor cm \rfloor$ such that no string in the set $w_x\{0,1\}^*$ has been queried in step (2) and $w_x \neq w_{x'}$ for any $w_{x'}$ defined earlier in this step. (Such a w_x must exist, if we choose n large enough.) Assume that $x = \langle \vec{y}, \vec{z}\rangle$ and let $j = t(m) - m - |w_x| + 1$. Let $B = B \cup \{w_x 0^j y : y \in \vec{y}\}$ and $B' = B' \cup \{w_x 0^j z : z \in \vec{z}\}$.

(5) For all x, $|x| \leq e_i =_{\text{defn}} \max\{q_i(n), 4t(n)\}$ and $x \in EL_m$, $m \geq n$, if $u = f(x)$ was not queried in step (2), then do the same as in step (1).

End of Stage i.

CLAIM 1. *If $x \in EL_m$, $m < n$, then $f(x) \in A$ if and only if $x \in LT^A[an, t(n)]$.*

Proof. After $f(x)$ is determined to be in A or $\overline{A}$, we add, later in step (1) and in steps (2) and (4), only strings of length longer than $t(m)$ to A. Thus whether $x \in LT^A[an, t(n)]$ is not affected by these actions.

In step (3), we may have added a string x_0 of length $3n$ to A, where x_0 is 1-random relative to B_0. We claim that if $|\alpha| \leq an$ and $|u| \leq n$, then $V^{B_0}(\alpha, u)$ cannot query x_0 in $t(n)$ moves. Suppose otherwise that it queries x_0 at the mth move, $m \leq t(n)$. Consider the following machine T_5: $T_5^{B_0}(\langle m, \alpha, u\rangle)$ simulates $V^{B_0}(\alpha, u)$ until the mth move; if the mth move is a query "$v \in ?B_0$," then prints v else prints 0.

Note that T_5 prints x_0 using a string of length

$$|\langle m, \alpha, u\rangle| = 2|m| + 2|\alpha| + |u| + 4 \leq 2\log(t(n)) + 2an + n + 6 < 3n - c_{T_5},$$

if n is chosen large enough. So this would imply that x_0 is not 1-random relative to B_0 (a contradiction).

The above shows that adding x_0 to A does not change the membership of x in $LT^A[an, t(n)]$. So Claim 1 is proven. □

CLAIM 2. *If $x = \langle \vec{y}, \vec{z}\rangle \in EL_m$, $m \geq n$, and $u = f(x)$ is queried and answered* YES *in step* (2), *then $x \in LT^A[bn, r(n)]$.*

Proof. In step (4), we found a string w_x of length $\lfloor cm \rfloor$ such that $w_x 0^j y \in A$ if and only if $y \in \vec{y}$, where $j = t(m) - m - |w_x| + 1$. So machine R_1 of Definition 2.7 is consistent with $\langle \vec{y}, \vec{z}\rangle$; i.e., $R_1^A(\langle j, w_x\rangle, y)$ accepts if $y \in \vec{y}$, and $R_1^A(\langle j, w_x\rangle, z)$ rejects if $z \in \vec{z}$. This implies that

$$LT_{R_1}^A(\langle \vec{y}, \vec{z}\rangle, r_1(t(m) + 1)) \leq 2|j| + |w_x| + 2 \leq 2\log(t(m)) + cm + 2 \leq bm - d_{R_1},$$

if n is chosen sufficiently large so that the above inequality holds for all $m \geq n$. Or,

$$LT^A(\langle \vec{y}, \vec{z}\rangle, p_1(r_1(t(m) + 1))) \leq LT^A(\langle \vec{y}, \vec{z}\rangle, r(m)) \leq bm.$$

This completes the proof of Claim 2, as well as the theorem. □

Remarks. (1) The reason that we need $r(n) \geq r_0(t(n) + 1)$ is that we cannot, in step (4), encode the information about $x = \langle \vec{y}, \vec{z}\rangle$ by short strings of the form $w_x y$, $y \in \vec{y}$. This is because we cannot guarantee that simulations in step (1), e.g., of $V^B(\alpha, u)$, did not query about $w_x y$, even if w_x is 1-random relative to B_0: the machine may use the second input u, which may have length $m = n - 1$, to generate w_x, which may be of length shorter than n.

(2) Both Theorems 3.3 and 3.4 can be generalized to sets $LT^A[s(n), t(n)]$ with size bound $s(n) > n$. For instance, Theorems 3.3 and 3.4 hold for classes $LT^A[an^2, t(n)]$ and $\overline{LT^A[bn^2, r(n)]}$.

4. NP-completeness. It has been pointed out in §1 that sets like *MINKT* or $KT[(n/2), n^3]$ do not seem to possess the properties of self-reducibility or paddability that are shared by most known NP-complete problems. This leads us to the conjecture that they are indeed not NP-complete (under the $\leq_m^P$-reduction). In this section, we show that the problem $MINKT^A$, under a slightly modified form where all time bounds t in inputs must be greater than or equal to $t_0(|x|)$, is neither in $co\text{-}NP^A$ nor $\leq_T^{P,A}$-complete for NP^A. This result supports our conjecture above.

On the other hand, we show that the problem $MINKT^A$, under the same modification, could be complete for NP^A under the weaker $\leq_T^{SNP,A}$-reduction. For the reader who is familiar with the notion of the high hierarchy [22], recall that the first two levels of the high hierarchy are

$$H_0^P = \{B \in NP : B \text{ is } \leq_T^P\text{- complete for } NP\},$$
$$H_1^P = \{B \in NP : B \text{ is } \leq_T^{SNP}\text{- complete for } NP\}.$$

It was proved in Ko [12] that $H_1^{P,A} \neq H_0^{P,A}$ relative to some oracle A. Our results here suggest $MINKT$ as a candidate of natural problems in $H_1^P - H_0^P$.

In the following, we let $MINKT_1^A = \{\langle x, 0^s, 0^t\rangle \in MINKT^A : t \geq t_0(|x|)\}$, and show that $MINKT_1^A$ is not complete for NP^A under the $\leq_T^{P,A}$-reduction. We note that this problem is essentially identical to the problem $MINKT^A$ since our main concern about $MINKT^A$ is the size bound rather than the lower level time bound. This also implies that $KT^A[an, t(n)]$ is not complete for NP^A.

THEOREM 4.1. *There exists a set A such that* $MINKT_1^A \notin co\text{-}NP^A$ *and* $MINKT_1^A$ *is not* $\leq_T^{P,A}$*-complete for* NP^A.[1]

Proof. Let $L_A = \{0^n : (\exists x, |x| = n)\ x \in A\}$. It is clear that $L_A \in NP^A$. Let $\{M_i\}$ be an enumeration of polynomial-time deterministic oracle TMs, and $\{N_i\}$ an enumeration of polynomial-time nondeterministic oracle TMs. Assume that q_i is a polynomial bounding the runtime of machines M_i and N_i. We need to construct set A to satisfy the following requirements:

$$R_{0,i} : (\exists x = x_i)\ [N_i^A(x) \text{ accepts } \iff x \in MINKT_1^A],$$

$$R_{1,i} : (\exists n = n_i)[0^n \in L_A \iff 0^n \notin L(M_i, A \oplus MINKT_1^A)].$$

Requirements $R_{0,i}$ are easy to satisfy, like those in Theorem 3.1. We only give a short sketch.

At Stage $2i$, we satisfy requirement $R_{0,i}$. Assume that sets A_{2i-1}, A'_{2i-1} and integer e_{2i-1} have been determined in the last stage. We choose a sufficiently large $n = n_i > e_{2i-1}$. Let z be a string of length n that is 1-random relative to A_{i-1}. Define $x = \langle z, 0^{n/2}, 0^{t_0(n)}\rangle$.

Then we simulate $N_i(x)$ using oracle A_{2i-1}. If $N_i^{A_{2i-1}}(x)$ rejects, then we do nothing. Otherwise, we fix an accepting path π of $N_i^{A_{2i-1}}(x)$ and find a string w of length $n/3$ that is 1-random relative to A_{2i-1} and such that no string in the set $w\{0,1\}^*$ has been queried in the path π. Let $A_{2i} = A_{2i-1} \cup \{w01^j : \text{the } j\text{th bit of } z \text{ is } 1\}$ and $A'_{2i} = A'_{2i-1} \cup \{w01^j : \text{the } j\text{th bit of } z \text{ is } 0\}$.

In the case of $N_i^{A_{2i-1}}(x)$ rejecting, it is easy to see that $x = \langle z, 0^{n/2}, 0^{t_0(n)}\rangle \notin MINKT_1^A$. If $N_i^{A_{2i-1}}(x)$ accepts, then $z = \xi_A(n, w)$ and so $\langle z, 0^{n/2}, 0^{t_0(n)}\rangle \in MINKT_1^A$, as long as we choose n large enough so that $|\langle n, w\rangle| = n/3 + 2\log n + 2 \leq n/2 - c_{T_1}$.

Next, we describe how to satisfy the second type of requirements $R_{1,i}$. In Stage $2i + 1$, we satisfy requirement $R_{1,i}$. Again, we assume that A_{2i}, A'_{2i}, and e_{2i} have been defined in the last stage. Select a sufficiently large n. Let $B = A_{2i}$, $B' = A'_{2i}$. Simulate M_i on input 0^n. In the simulation, we may encounter two types of queries: the first type querying to the oracle A and the second type querying to the oracle $MINKT_1^A$. We simulate these queries as follows:

(1) For the query "$y \in ?A$" of the first type, answer YES if $y \in B$ and answer NO otherwise. For each y answered NO, if y was not in B' then add y to B'.

(2) Assume that all the queries "$y = \langle z, 0^s, 0^t\rangle \in ?MINKT_1^A$" of the second type have $t \geq t_0(|z|)$ (otherwise just answer NO). For such a query, simulate $U^A(\alpha)$, for all α of length less than or equal to s, each for t moves. In these simulations, for each query "$w \in ?A$," we answer YES if $w \in B$, answer NO if $w \in B'$ or $|w| = n$ (but do not add those w of length n to B' yet), and consider both answers in the simulation

[1] This result has been independently proved by Professor Juris Hartmanis (unpublished).

otherwise. Thus, for each α of length less than or equal to s, we obtain a computation tree C_α of $U^A(\alpha)$. The tree C_α has depth less than or equal to t.

(2.1) If there exists a tree C_α such that one of its computation paths prints z, then fix this tree C_α and this path π. For each query w made in this path, if it is answered YES then add it to B else add it to B'. In particular, for those w of length n in the path π (their answers were fixed to NO when building the tree), we also add them to B'. In this case, we answer YES to the query "$y \in ? MINKT_1^A$."

(2.2) If no C_α contains a path printing z, then do nothing, and answer NO to the query "$y \in ? MINKT_1^A$."

If the above simulation of $M_i^{A \oplus MINKT_1^A}(0^n)$ accepts, then let $A_{2i+1} = B$. Otherwise, we choose an x_0 of length n that is 1-random relative to B and $x_0 \notin B \cup B'$. Let $A_{2i+1} = B \cup \{x_0\}$. (Note that for each simulation of the second type query "$y = \langle z, 0^s, 0^t \rangle \in ? MINKT^{A}$" (step (2)), we add at most $t \leq |y| \leq q_i(n)$ strings into $B \cup B'$. We execute at most $q_i(n)$ times of steps (1) and (2), and so, in total, we add at most $(q_i(n))^2$ many strings to $B \cup B'$. Thus, by choosing n large enough, such an x_0 must exist.)

End of Stage $2i+1$.

CLAIM 1. *If the simulation of* $M_i(0^n)$ *as described in Stage* $2i+1$ *accepts, then* $M_i(0^n)$ *accepts with oracle* $A \oplus MINKT_1^A$.

Proof. All answers to queries of the first type "$y \in ? A$" in the simulation were made to be consistent with the sets B and B' at that time and their memberships in B or B' are never changed later. So these answers are all correct with respect to the oracle A.

For the queries "$y = \langle z, 0^s, 0^t \rangle \in ? MINKT^{A}$" of the second type, if it was answered YES, then we must have found a computation path π in a tree C_α for some α of length less than or equal to s and all answers in the path π to queries of the form "$w \in ? A$" were made consistent with B and B'. So the answer YES is always correct. If it is answered NO, then for any set D that is consistent with sets B and $B' \cup \{0,1\}^n$ at that time (i.e., $B \subseteq D$ and $B' \cap D = \{0,1\}^n \cap D = \emptyset$) the answer NO is still correct, because we have simulated all possible trees C_α relative to all such sets D. Since we did not add any string of length n to B later, the answer NO is correct relative to set A. □

From Claim 1 above, we see that if the simulation of $M_i(0^n)$ in Stage $2i+1$ accepts, then requirement $R_{1,i}$ is satisfied, because it is obvious that in this case $0^n \notin L_A$.

Next we consider the case where the simulation of $M_i(0^n)$ in Stage $2i+1$ rejects.

CLAIM 2. *Assume that the simulation of* $M_i(0^n)$ *in stage* $2i+1$ *rejects and that* $y = \langle z, 0^s, 0^t \rangle$ *was queried to oracle* $MINKT_1^A$ *in the simulation and was answered* NO. *Then, it must be true that* $s < n/2$.

Proof. Suppose, by way of contradiction, that $s \geq n/2$. Then, $|z| \geq n/3$ (otherwise, we could print z using z itself as a program that has size $< n/2$;[2] this contradicts the assumption that y is answered NO). As noted above, we added, in the simulation of $M_i(0^n)$ in Stage $2i+1$, at most $(q_i(n))^2$ many strings to sets B and B'. There must exist some string w of length $n/4 > 2\log(q_i(n))$ such that $w\{0,1\}^* \cap (B \cup B') = \emptyset$ when y was queried. Let $D = B \cup \{w0^{3n/4}1^j : \text{the } j\text{th bit of } z \text{ is } 1\}$. Then D is consistent with B and $B' \cup \{0,1\}^n$. Also, $T_0^D(\langle |z|, 3n/4, w \rangle)$ prints z in $t_1(4|z|)$ moves (note that $|z| \geq n/3$). This implies that $U^D(\alpha)$ prints z in $t_0(|z|) \leq t$ moves for some program α of length less than or equal to $c_{T_0} + 2\log|z| + 2\log n + n/4 < n/2$, if n is sufficiently

[2] We assumed in §2 that $t_0(n)$ is large enough to do this.

large. In other words, one of the computation paths of C_α that corresponds to the oracle D prints z. This contradicts our assumption that "$y \in ? MINKT_1^A$" receives the answer NO. □

CLAIM 3. *Assume the same as in Claim 2. Then, x_0 is never queried in the computation of $U^A(\alpha)$ in t moves for all α of length less than or equal to s.*

Proof. Assume otherwise that $U^A(\alpha)$ queries about x_0 in the mth move, $|\alpha| \leq s < n/2$ and $m \leq t$. Let B be the set defined after we completed the simulation of $M_i(0^n)$ in Stage $2i+1$, then the computation of $U^B(\alpha)$ is the same as $U^A(\alpha)$ in the first $m-1$ moves. Consider the machine T_2 defined in §3. $T_2^A(\langle m, \alpha \rangle)$ prints x_0 with a program of length $|\langle m, \alpha \rangle| \leq 2\log(q_i(n)) + n/2 + 2 \leq n - 1 - c_{T_2}$. From our choice of n, this contradicts the 1-randomness of x_0 relative to B. □

CLAIM 4. *If the simulation of $M_i(0^n)$ in Stage $2i+1$ rejects, then $M_i(0^n)$ rejects relative to $A \oplus MINKT_1^A$.*

Proof. The proof of Claim 4 is almost identical to Claim 1. We only need to add that, from Claim 3, when a second type query "$y = \langle z, 0^s, 0^t \rangle \in ? MINKT_1^A$" was answered NO, none of its simulation tree C_α queries about x_0 and so the simulated answer remains correct. □

Claim 4 above establishes that if the simulation of $M_i(0^n)$ in Stage $2i+1$ rejects, then requirement $R_{1,i}$ is satisfied. □

Next we show that the set $KT^A[an, t(n)]$ is $\leq_T^{SNP,A}$-complete for NP^A if $0 < a < 1$ and $t(n) \geq t_0(n)$. This also implies that $MINKT_1^A$ is $\leq_T^{SNP,A}$-complete for NP^A. Note that if $NP^A = co\text{-}NP^A$, then this is trivial. We construct set A such that this condition holds while $NP^A \neq co\text{-}NP^A$.

THEOREM 4.2. *Let $0 < a < 1$ and t be a polynomial such that $t(n) \geq t_0(n)$. There exists a set A such that $NP^A \neq co\text{-}NP^A$ and $KT^A[an, t(n)]$ is $\leq_T^{SNP,A}$-complete for NP^A.*

Proof. Let $t'(n) = t(n^2)$. Let $L_A = \{0^n : (\exists x, |x| = 2t'(n))\ 0x \in A\}$. Then, it is clear that $L_A \in NP^A$. Also let Q_A be a set $\leq_m^{P,A}$-complete for NP^A such that for all x the question of whether $x \in Q_A$ depends only on $A^{\leq |x|}$. (For instance, let Q_A be the generic complete set consisting of all $\langle x, i, 0^j \rangle$ such that the ith nondeterministic oracle machine N_i accepts x in j moves, with oracle A.)

We are going to define a function $f_A(x,y)$ that is computable in polynomial time relative to A such that the following requirements are satisfied for all $x \in \{0,1\}^*$ and all integers $i \geq 0$:

$$R_{0,x} : x \notin Q_A \iff (\exists y, |y| = |x|)\ f_A(x,y) \notin KT^A[an, t(n)]$$

$$R_{1,i} : (\exists n = n_i)\ [N_i^A(0^n) \text{ accepts } \iff 0^n \in L_A].$$

Requirements $R_{0,x}$ imply that $\overline{Q_A} \in NP(A \oplus KT^A[an, t(n)])$ and so $KT^A[an, t(n)]$ is $\leq_T^{SNP,A}$-complete for NP^A. Requirements $R_{1,i}$ imply that $NP^A \neq co\text{-}NP^A$.

We first define a sequence of integers: $\ell_0 = 1$, $\ell_{i+1} = 2^{2^{\ell_i}}$. We partition the set $\{0,1\}^*$ into two groups: $G_0 = \{x : \log \ell_i \leq |x| < \ell_i \text{ for some } i \geq 1\}$ and $G_1 = \{x : \ell_i \leq |x| < 2^{\ell_i} \text{ for some } i \geq 1\}$. We define the function f_A as follows:

$$f_A(x,y) = \begin{cases} \xi_A(|x|, 1xy0^{t'(|x|)}) & \text{if } x \in G_0, \\ \xi_A(|x|^2, 1xy0^{t'(|x|)}) & \text{if } x \in G_1. \end{cases}$$

We also let $S(x,y) = \{1xy0^{t'(|x|)}01^j : 1 \leq j \leq |x|\}$ if $x \in G_0$, and $S(x,y) = \{1xy0^{t'(|x|)}01^j : 1 \leq j \leq |x|^2\}$ if $x \in G_1$.

At Stage i, we find the least j such that $\ell_j > e_{i-1}$, and (a) $(1-a)n > 4\log(q_i(n)) + 4\log(t'(n)) + 6 + c_{T_3}$, and (b) $an^2 > 2n + 4\log n + 4 + 2c_{T_1}$, for all $n \geq \ell_j$. (Recall that machine T_1 was defined in Definition 2.6, and machine T_3 was defined in §3.) Let $n = \ell_j$ and $B = A_{i-1}$, $B' = A'_{i-1}$.

(1) For all x, $e_{i-1} \leq |x| < n$, do the following in increasing order: Determine whether $x \in Q_B$. If $x \in Q_B$, then define

$$B' = B' \cup \Big(\bigcup_{|y|=|x|} S(x,y) \Big).$$

(Thus, for all y, $|y| = |x|$, $f_A(x,y) = 0^{|x|}$ if $x \in G_0$ and $f_A(x,y) = 0^{|x|^2}$ if $x \in G_1$. So, for all y, $|y| = |x|$, $f_A(x,y) \in KT^B[an, t(n)]$.) If $x \notin Q_B$, then choose a string w of length $|x|$ if $x \in G_0$, or of length $|x|^2$ if $x \in G_1$, that is 1-random relative to B. Define

$$B = B \cup \{1xy0^{t'(|x|)}01^j \in S(x,y) : |y| = |x|, \text{ the } j\text{th bit of } w \text{ is } 1\},$$
$$B' = B' \cup \{1xy0^{t'(|x|)}01^j \in S(x,y) : |y| = |x|, \text{ the } j\text{th bit of } w \text{ is } 0\}.$$

(Thus, for all y, $|y| = |x|$, $f_A(x,y) = w$.)

When all these are done, let $B_0 = B$ and $B'_0 = B'$.

(2) We simulate N_i on 0^n with queries "$w \in ?A$" answered as follows:

(2.a) If $w \in B$ then answer YES; if $w \in B'$ then answer NO.

(2.b) If $w \notin B \cup B'$ and $w \notin S(x,y)$ for any x, y, $|x| = |y|$, then answer NO (but do not add w to B').

(2.c) If $w \notin B \cup B'$ and $w \in S(x,y)$ for some x, y, $|x| = |y|$, then consider both answers in the simulation.

We obtain from the above simulation a computation tree C of $N_i(0^n)$.

(3) If one of the computation paths in C accepts, we fix this path π and do the following:

(3.1) Add all strings queried and answered YES in the path π to B, and add all strings queried and answered NO in the path π to B' (including those answered NO in case (2.b)).

(3.2) Choose the least x_0, $|x_0| = 2t'(n)$, such that $0x_0 \notin B'$ and add $0x_0$ to set B.

(3.3) For each pair (x,y), $|x| = |y| \geq n$, such that some $w \in S(x,y)$ has been queried in the path π (note then $x \in G_1$), do the following: First, for all strings $w' \in S(x,y)$ that were not queried in the path π, add w' to B'. Then, choose a string z of length n such that (1) z is 1-random relative to B_0, (2) no string in $1xz\{0,1\}^*$ has been queried in the path π, and (3) z is different from all other z' defined in this step from other pairs of (x', y'). (Note that there are at most $q_i(n)$ pairs of (x,y) for which $S(x,y)$ contains some string queried in path π. So, from our choice of n, such strings z must exist.) Define

$$B = B \cup \{1xz01^j : 1xy0^{t'(|x|)}01^j \in B\},$$
$$B' = B' \cup \{1xz01^j : 1xy0^{t'(|x|)}01^j \in B'\}.$$

(Thus, $\xi_A(|x|^2, 1xz) = f_A(x,y)$.)

(4) If none of the computation paths of C accepts, then do nothing.

(5) For each x, $n < |x| \leq 2^n$, do the following in increasing order: Let $Y_x = \{y : |y| = |x| \text{ and } S(x,y) \cap (B \cup B') = \emptyset\}$. Determine whether $x \in Q_B$. If $x \in Q_B$, then let

$$B' = B' \cup \Big(\bigcup_{y \in Y_x} S(x,y) \Big).$$

(Thus, for all $y \in Y_x$, $f_A(x,y) = 0^{|x|^2}$ and so $f_A(x,y) \in KT^B[an, t(n)]$.) If $x \notin Q_B$, then choose a string w of length $|x|^2$ that is 1-random relative to B and define

$$B = B \cup \{1xy0^{t'(|x|)}01^j \in S(x,y) : y \in Y_x, \text{ the } j\text{th bit of } w \text{ is } 1\},$$

$$B' = B' \cup \{1xy0^{t'(|x|)}01^j \in S(x,y) : y \in Y_x, \text{ the } j\text{th bit of } w \text{ is } 0\}.$$

(Thus, for all $y \in Y_x$, $f_A(x,y) = w$.)

(6) Let $A_i = B$, $A'_i = B'$, and $e_i = 2^n$.

End of Stage i.

Let $A = \cup_{i=0}^{\infty} A_i$. In the following, consider Stage i.

CLAIM 1. *If $e_{i-1} \leq |x| < n$, then $x \in Q_A \iff (\forall y, |y| = |x|)\ f_A(x,y) \in KT^A[an, t(n)]$.*

Proof. First we observe that in Stage i, after we determine whether $x \in Q_B$ relative to the set B at that time, we never add any string shorter than $|x|$ to A. So the simulation of $x \in Q_B$ is always correct.

Next, we observe that if $x \in Q_A$, then we defined $f_A(x,y) = 0^{|x|}$ or $0^{|x|^2}$ and hence $f_A(x,y) \in KT^A[an, t(n)]$ for all y, $|y| = |x|$, since 0^k can be printed from a program of size $O(\log k)$ in $t(k)$ moves (see the assumption in Definition 2.6). If $x \notin Q_A$, then we defined $f_A(x,y) = w$ for all y, $|y| = |x|$, where w is of length $|x|$ or $|x|^2$, depending on whether $x \in G_0$ or $x \in G_1$, and is 1-random relative to B. We note that $|w| < n$ (if $x \in G_0$, then $|w| = |x| < n$; if $x \in G_1$, then $|x| < \log n$ and so $|w| = |x|^2 < n$). Since $B_0 - B$ contains no string of length less than or equal to $t'(|x|)$, $KT^{B_0}(w, t'(|x|)) \geq |w| - 1$.

Now consider the strings in $A_i - B_0$. The only strings of length less than or equal to $t'(|x|)$ are those added to A in step (3.3), of the form $u = 1x'z01^j$, $|x'| \geq n$, $|z| = n$, $1 \leq j \leq |x'|^2$, z being 1-random relative to B_0. We show that for any program α of length less than or equal to $a|w|$, $U^A(\alpha)$ cannot query about such a string in $t'(|x|)$ moves. Suppose otherwise that $U^A(\alpha)$, $|\alpha| \leq a|w|$, first queries about such a string $u = 1x'z01^j$ at the mth move, $m \leq t'(|x|)$. Then, in the first $m-1$ moves, $U^A(\alpha)$ and $U^{B_0}(\alpha)$ behave the same because $(A - B_0)^{\leq t'(|x|)}$ contains only strings of this form. Now let $i_1 = |x'| + 1$, $i_2 = i_1 + n - 1$. Then, $T_3^{B_0}(\langle i_1, i_2, m, \alpha\rangle)$ prints z, where T_3 is the machine defined in §3, before Theorem 3.2. Therefore, $K^{B_0}(z) \leq 2|i_1| + 2|i_2| + 2|m| + |\alpha| + 6 + c_{T_3} \leq 4\log(q_i(n)) + 2\log(t'(n)) + an + 6 + c_{T_3} \leq n - 1$. This is a contradiction.

The above shows that all strings added in $A_i - B_0$ cannot affect the computation of $U^{B_0}(\alpha)$, $|\alpha| \leq a|w|$, in $t'(|x|)$ moves. So $U^A(\alpha)$ does not print w for all α of length less than or equal to $a|w|$ in $t'(|x|)$ moves. Since $t'(|x|) \geq t(|w|)$, we conclude that $f_A(x,y) = w \notin KT^A[an, t(n)]$ for all y, $|y| = |x|$. So the requirement $R_{0,x}$ is satisfied. □

CLAIM 2. *Requirement $R_{1,i}$ is satisfied in Stage i.*

Proof. First, if the simulation of $N_i(0^n)$ in Stage i accepts, then all answers to the queries made in an accepting path π have been made correct with respect to the oracle A. So, in this case, $N_i^A(0^n)$ accepts. Also, a string $0x_0$ of length $2't(n) + 1$ is added to A and so $0^n \in L_A$.

If the simulation of $N_i(0^n)$ in Stage i rejects, then only strings added to A later in Stage i are in $S(x,y)$ for some x, y, $|x| = |y| \geq n$. We observe that $N_i^A(0^n)$ must reject; otherwise, there would be an accepting path in the computation tree C constructed in step (2) and we would have made A consistent with this path and the simulation would accept. In addition, it is clear that no string in $0\{0,1\}^{2t'(n)}$ is added to A later and so $0^n \notin L_A$. □

CLAIM 3. *If* $n \leq |x| < 2^n$, *then* $x \in Q_A \iff (\forall y, |y| = |x|)\ f_A(x,y) \in KT^A[an, t(n)]$.

Proof. Again, the simulation of whether $x \in Q_A$, performed in Stage i, must be correct because we do not add any string of length less than or equal to $|x|$ to A after the simulation.

Assume that $x \in Q_A$ and let B_x and B'_x be the sets B and B' defined when the query "$x \in ?Q_A$" was considered in Stage i. Then for each y, $|y| = |x|$, either $S(x,y) \cap (B_x \cup B'_x) = \emptyset$ (i.e., $y \in Y_x$) or $S(x,y) \subseteq B_x \cup B'_x$. In the former case, we made $f_A(x,y) = 0^{|x|^2}$ and so it is in $KT^A[an, t(n)]$. In the latter case, we note that $f_A(x,y) = \xi_A(|x|^2, 1xz)$ for some z of length n. So, $f_A(x,y)$ can be printed by machine T_1 of Definition 2.6 from a program of size less than or equal to $4\log|x| + 2|x| + 3$ in $t_1(|x|^2)$ moves. That is, it has KT-complexity

$$KT^A(f_A(x,y), t(|x|^2)) \leq 2|x| + 4\log|x| + 2 + c_{T_1} \leq a|x|^2.$$

Or, $f_A(x,y) \in KT^A[an, t(n)]$.

Next assume that $x \notin Q_A$. Consider two cases. First, if the simulation of $N_i(0^n)$ rejects, then for all y, $S(x,y) \cap (B \cup B') = \emptyset$. Therefore, for all y, $f_A(x,y) = w$, a 1-random string relative to B at that time. Since no string of length less than or equal to $t(|w|)$ was added to A later in the stage, $KT^A(w, t(|w|)) \geq n - 1 \geq an$. So for all y, $|y| = |x|$, $f_A(x,y) \notin KT^A[an, t(n)]$.

In the second case, if the simulation of $N_i(0^n)$ accepts, then for at most $q_i(n)$ many y, $|y| = |x|$, $S(x,y) \cap (B \cup B') \neq \emptyset$. So, by our choice of n, there must be at least one y for which $S(x,y)$ was untouched. Similar to the first case, for these y, $f_A(x,y)$ was made to be 1-random relative to B, and hence 1-random relative to A. The requirement $R_{0,x}$ is satisfied. □

Remark. Note that, in the above proof, if $x \notin Q_A$, then for at least half of y, $|y| = |x|$, $f_A(x,y) \notin KT^A[an, t(n)]$, as long as we choose n_i big enough. This implies that set $KT^A[an, t(n)]$ is actually $\leq_T^{coRP,A}$-complete in the sense that every set $B \in co\text{-}NP^A$ is actually in $RP(A \oplus KT^A[an, t(n)])$. If we exchange the roles of Q_A and $\overline{Q_A}$, then the proof remains valid, and so $KT^A[an, t(n)]$ is $\leq_T^{RP,A}$-complete for NP^A. Note however that this result is meaningful only if we know that $NP^A \neq R^A$. Indeed, our proof can be modified so that $NP^A \neq R^A$ and $KT^A[an, t(n)]$ is $\leq_T^{RP,A}$-complete for NP^A, but the proof involves extra work. We omit it here.

Next we consider the completeness of the problem $MINLT^A$. Let

$$MINLT_1^A = \{\langle\langle\vec{y}, \vec{z}\rangle, 0^s, 0^t\rangle \in MINLT^A : t \geq r_0(|\langle\vec{y}, \vec{z}\rangle|)\}.$$

THEOREM 4.3. *There exists a set* A *such that* $MINLT_1^A \notin co\text{-}NP^A$ *and* $MINLT_1^A$ *is not* $\leq_T^{P,A}$*-complete for* NP^A.

Proof. The proof is similar to that of Theorem 4.1. It consists of two sets of diagonalization requirements. The first is for the condition $MINLT_1^A \notin co\text{-}NP^A$. It is proved by a simple diagonalization, as in Theorem 3.3. We omit it here. The second is for the condition $MINKT_1^A$ not $\leq_T^{P,A}$-complete for NP^A. We sketch in the following the modification of the proof in Theorem 4.1 for the problem $MINLT_1^A$. Our requirements are

$$R_{1,i} : (\exists n = n_i)[0^n \in L_A \iff 0^n \notin L(M_i, A \oplus MINLT_1^A)],$$

where $L_A = \{0^n : (\exists x, |x| = n)\ x \in A\}$.

At Stage $2i+1$, we satisfy requirement $R_{1,i}$. Let e_{2i}, A_{2i}, and A'_{2i} be defined as in Theorem 4.1. Select a sufficiently large integer n. Let $B = A_{2i}$ and $B' = A'_{2i}$. Simulate $M_i(0^n)$, with the queries answered as follows:

(1) For the query "$w \in ?A$" of the first type, answer YES if $w \in B$, and answer NO if $w \notin B$. Add w to B' if w was not in $B \cup B'$ and w is answered NO.

(2) For the query "$\langle\langle\vec{y},\vec{z}\rangle, 0^s, 0^t\rangle \in ?MINLT_1^A$" of the second type (assuming that $t \geq r_0(|\langle\vec{y},\vec{z}\rangle|)$), we simulate V^A on all programs α, $|\alpha| \leq s$, on all $y \in \vec{y}$ and all $z \in \vec{z}$, each for t moves. In the simulation, for each query "$w \in ?A$", we answer YES if $w \in B$ and answer NO if $w \in B'$ or $|w| = n$. Otherwise, we consider both answers. Thus, for each α, we get trees $C_{\alpha,u}$, for each $u \in \vec{y} \cup \vec{z}$. Each tree $C_{\alpha,u}$ is of depth less than or equal to t. Let $C^X_{\alpha,u}$ denote the computation path in the tree $C_{\alpha,u}$ that is defined by oracle X.

(2.1) If there exists a program α and an oracle D such that $B \subseteq D$, $B' \cup \{0,1\}^n \subseteq \overline{D}$, and $C^D_{\alpha,y}$ accepts for all $y \in \vec{y}$, and $C^D_{\alpha,z}$ rejects for all $z \in \vec{z}$, then fix this program α and the set D. For each query w made in the paths $C^D_{\alpha,u}$, $u \in \vec{y} \cup \vec{z}$, add w to B if $w \in D$ and add w to B' if $w \notin D$. We answer YES to the query "$\langle\langle\vec{y},\vec{z}\rangle, 0^s, 0^t\rangle \in ?MINLT_1^A$." (Note that there are at most $q_i(n)$ many trees $C_{\alpha,u}$ and each has depth $\leq t \leq q_i(n)$. So, we add at most $(q_i(n))^2$ many w to $B \cup B'$ for each query of the second type.)

(2.2) If the condition specified in (2.1) fails, then answer NO to the query "$\langle\langle\vec{y},\vec{z}\rangle, 0^s, 0^t\rangle \in ?MINLT_1^A$."

If the above simulation of $M_i(0^n)$ accepts, then let $A_{2i+1} = B$. Otherwise, choose x_0 of length n such that $x_0 \notin B \cup B'$ and x_0 is 1-random relative to B. Let $A_{2i+1} = B \cup \{x_0\}$. (Note that for each query of the second type, we add at most $(q_i(n))^2$ strings to $B \cup B'$. Since the total number of queries (of the first type or of the second type) is at most $q_i(n)$, we know that $B \cup B' - A_{2i}$ contains at most $(q_i(n))^3$ strings. So, such an x_0 exists as long as $n > 3\log(q_i(n))$.)

End of Stage $2i+1$.

The proofs of the following three claims are very similar to those of Theorem 4.1 and are omitted here.

CLAIM 1. *If the above simulation of $M_i(0^n)$ accepts, then $M_i(0^n)$ accepts with respect to the oracle $A \oplus MINLT_1^A$.*

CLAIM 2. *Assume that the simulation of $M_i(0^n)$ of Stage $2i+1$ rejects and that $x = \langle\langle\vec{y},\vec{z}\rangle, 0^s, 0^t\rangle$ was queried to the oracle $MINLT_1^A$ in the simulation and answered NO. Then, $s < n/2$.*

CLAIM 3. *Assume the same as in Claim 2. Then, for any α, $|\alpha| \leq s$, and any $u \in \vec{y} \cup \vec{z}$, $C^A_{\alpha,u}$ does not query about x_0.*

From Claim 3, we see that adding x_0 to A does not change the NO answers to the second type queries. So the simulation of $M_i(0^n)$ is correct with respect to the oracle $A \oplus MINLT_1^A$. This completes the proof. □

THEOREM 4.4. *Let $0 < a < 1$ and let $t(n)$ be a polynomial such that $t(n) \geq r_0(n)$ for all n. There exists an oracle A such that $NP^A \neq co\text{-}NP^A$ and the set $LT^A[an, t(n)]$ is $\leq_T^{SNP,A}$-complete for NP^A.*

Proof. Again, the proof is similar to Theorem 4.2. We only give a sketch. First recall that σ_j^m is the m-bit binary representation of integer j, and that machine $T_4^A(\langle m,\alpha\rangle)$ simulates $V^A(\alpha,\sigma_j^m)$ for $j = 1,\cdots,m$, and prints the outcomes in increasing order. We assume that if $V^A(\alpha,\sigma_j^m)$ halts in time $r(m)$ for all $j = 1,\cdots,m$, then $T_4^A(\langle m,\alpha\rangle)$ halts in time $t_4(r(m))$. Let $t'(n) = p_0(t_4(t(n^2)))$.

Let Q_A be an NP^A-complete set as defined in Theorem 4.2. For each pair (x,y),

$|x| = |y|$, let $f_A(x,y)$ and $S(x,y)$ be defined as in Theorem 4.2 (with respect to the new $t'(n)$). Also define $\vec{u} = \{\sigma_j^m$: the jth bit of $f_A(x,y)$ is 1$\}$ and $\vec{v} = \{\sigma_j^m$: the jth bit of $f_A(x,y)$ is 0$\}$, where $m = |x|$ if $x \in G_0$ and $m = |x|^2$ if $x \in G_1$. Let $g_A(x,y) = \langle \vec{u}, \vec{v} \rangle$. Our requirements are, for *almost* all $x \in \{0,1\}^*$ (such that $(1-a)|x| > 2\log|x| + c_0$ for some constant c_0) and all integers i,

$$R_{0,x} : x \notin Q_A \iff (\exists y, |y| = |x|)\ g_A(x,y) \notin LT^A[an, t(n)],$$

$$R_{1,i} : \text{ same as } R_{1,i} \text{ in Theorem 4.2.}$$

Assume the same setting for Stage i as in Theorem 4.2. We choose a sufficiently large $\ell_j > e_{i-1}$ and let $n = \ell_j$. We do the same as in Stage i of Theorem 4.2, with the following modification:

(3.3) For each pair (x,y) such that $w \in S(x,y)$ has been queried in the path π, do the following: First, for all strings $w' \in S(x,y)$ that were not queried in the path π, add w' to B'. Then, choose a string z of length $4n$ such that (1) z is 1-random relative to B_0 and (2) no string in $z\{0,1\}^*$ has been queried in the path π. Define

$$B = B \cup \{zxy\sigma_j^{|x|^2} : 1xy0^{t'(|x|)}01^j \in B\},$$

$$B' = B' \cup \{zxy\sigma_j^{|x|^2} : 1xy0^{t'(|x|)}01^j \in B'\}.$$

It is obvious that requirement $R_{1,i}$ is satisfied in Stage i.

CLAIM 1. *If* $e_{i-1} \le |x| < n$ *then* $x \in Q_A \iff (\forall y, |y| = |x|)\ g_A(x,y) \in LT^A[an, t(n)]$.

Proof. It is clear that the simulation of whether $x \in Q_A$ is always correct.

If $x \in Q_A$, then for all y, $|y| = |x|$, $f_A(x,y) = 0^{|x|}$ or $0^{|x|^2}$, and so $g_A(x,y) = \langle \lambda, \vec{v} \rangle$, where $\vec{v} = \{\sigma_j^{|x|} : 1 \le j \le |x|\}$ if $x \in G_0$, or $\vec{v} = \{\sigma_j^{|x|^2} : 1 \le j \le |x|^2\}$ if $x \in G_1$. It is clear that $g_A(x,y) \in LT^A[an, t(n)]$, because $g_A(x,y)$ is recognized by an "always rejecting" program of a constant size in $r_0(n) \le t(n)$ moves.

If $x \notin Q_A$, we claim that $KT^A(w, t'(|x|)) \ge |w| - 1$.

Case 1. $|x| < \log n$. Then for all y, $|y| = |x|$, if $f_A(x,y) = w$, then $|w| = |x|^2 < \log^2 n$. When n is chosen large, no string added in $B_0 - B$ or in step (3.3) is less than or equal to $t'(|x|)$ and so $KT^A(w, t'(|x|)) \ge |w| - 1$.

Case 2. $|x| \ge \log n$. Then $x \in G_0$ and so for all y, $|y| = |x|$, $f_A(x,y) = w$ has length equal to $|x| < n$, and $KT^{B_0}(w, t'(|x|)) \ge |w| - 1$, because $B_0 - B$ contains no string of length less than or equal to $t'(|x|)$. Only strings in $A - B_0$ that are of length less than or equal to $t'(|x|)$ are of the form $zx'y'\sigma_j^{m'}$, where z has length $4n$ and is 1-random relative to B_0. We observe that for any α, $|\alpha| \le |w|$, and any u, $|u| \le |w|$, $V^{B_0}(\alpha, u)$ cannot query about such a string $zx'y'\sigma_j^{m'}$ in $t'(|x|)$ moves, because $|\langle \alpha, u \rangle| \le 3n + 2$, but z is of length $4n$ and is 1-random relative to B_0. Therefore, $KT^A(w, t'(|x|)) \ge |w| - 1$.

Now suppose, by way of contradiction, that $g_A(x,y) = \langle \vec{u}, \vec{v} \rangle \in LT^A[an, t(n)]$ and so $V^A(\alpha, u)$ accepts for all $u \in \vec{u}$ and $V^A(\alpha, v)$ rejects for all $v \in \vec{v}$ for some α of length less than or equal to $a|w|$. Note that each string in $\vec{u} \cup \vec{v}$ has length equal to $|w|$ and so $V^A(\alpha, u)$ halts in $t(|w|)$ moves for all $u \in \vec{u} \cup \vec{v}$. Then, $T_4^A(\langle |w|, \alpha \rangle)$ simulates $V^A(\alpha, \sigma_j^{|w|})$, $j = 1, \cdots, |w|$, to print w in time $t_4(t(|w|)) \le t_4(t(|x|^2))$. We get

$$KT_{T_4}^A(w, t_4(t(|x|^2))) \le 2\log|w| + 2 + |\alpha| < |w| - c_0 + 2,$$

because we only consider strings x such that $(1-a)|x| > 2\log|x| + c_0$. Let $c_0 = c_{T_4} + 3$, we have $KT^A(w, t'(|x|)) < |w| - 1$. This contradicts our claim above. □

CLAIM 2. *If* $n \le |x| < 2^n$, *then* $x \in Q_A \iff (\forall y, |y| = |x|)\ g_A(x,y) \in LT^A[an, t(n)]$.

Proof. The proof is essentially the same as for Claim 1. We only point out that if $x \in Q_A$ and $S(x,y) \subseteq B_z \cup B'_x$, then in step (3.3) we have found some z, $|z| = 4n$, such that $zxy\sigma_j^{|x|^2} \in B$ if and only if the jth bit of $f_A(x,y)$ is 1. So, the machine R_1^A, with program zxy, is consistent with $g_A(x,y) = \langle \vec{u}, \vec{v} \rangle$: $R_1^A(0, zxy, \sigma_j^{|x|^2})$ accepts if and only if the jth bit of $f_A(x,y)$ is 1 if and only if $\sigma_j^{|x|^2} \in \vec{u}$. Thus,

$$LT^A(g_A(x,y), t(|x|^2)) \le 6|x| + d_{R_1} \le |x|^2/2,$$

if n is chosen sufficiently large (note $t(|x|^2) \ge r_0(|x|^2)$). □

5. P-immunity. In this section we investigate the property of P-immunity of the set $KT[an, t(n)]$ and its complement $\overline{KT[an, t(n)]}$, $0 < a < 1$. The set $KT[an, t(n)]$ contains all "easy" strings and so is immediately not P-immune. For instance, the set $KT[\log n, n^3]$ is in P and so is a witness that $KT[an, n^3]$ is not P-immune. On the other hand, it is difficult to construct an infinite subset in P of $\overline{KT[an, t(n)]}$, because it must consist of only hard strings and any constructive definition of a set in P is likely to produce some easy strings. For instance, if $t(n) = n^{\log n}$, then $\overline{KT[an, t(n)]}$ is $\mathcal{C}$-immune for $\mathcal{C} = \{A : A \text{ is } P\text{-printable}\}$.

PROPOSITION 5.1. *Let* $0 < a < 1$ *and* $t(n) = n^{\log n}$. *Then* $\overline{KT[an, t(n)]}$ *does not have an infinite subset that is P-printable.*

Sketch of Proof. Assume, by way of contradiction, that A is an infinite, P-printable subset of $\overline{KT[an, t(n)]}$. Assume that T_6 is a polynomial-time TM that prints the set $A \cap \{0,1\}^n$ from input 0^n. Choose a sufficiently large n such that $A \cap \{0,1\}^n \ne \emptyset$. Let x be the first string in the list printed by $T_6(0^n)$. Then, we can design another Turing machine T_7 that simulates $T_6(0^n)$ and prints the first string of the output of $T_6(0^n)$. It is clear that the size of T_7 is only a constant plus $\log n$ and hence x has low KT-complexity. This is a contradiction. □

In the following, we show that, in the relativized form, Proposition 5.1 cannot be generalized too much. In other words, we show that the set $\overline{KT^A[an, t(n)]}$ could be both P^A-immune or non-P^A-immune, depending on oracle sets A.

THEOREM 5.2. *Let* $0 < a < 1$ *and t be a polynomial such that* $t(n) \ge t_0(n)$. *There exists an oracle A such that* $\overline{KT^A[an, t(n)]}$ *is* NP^A*-immune.*

Proof. Let $\{N_i\}$ be an enumeration of all polynomial-time nondeterministic oracle TMs. Assume that the runtime of N_i is bounded by a polynomial q_i. We need to construct a set A such that for each i,

$$R_i: \ L(N_i, A) \subseteq \overline{KT^A[an, t(n)]} \text{ implies that } L(N_i, A) \text{ is finite.}$$

We construct set A in stages. In each Stage n, we try to satisfy the least unsatisfied requirement R_i by finding a witness x of length n such that $x \in L(N_i, A) \cap KT^A[an, t(n)]$. Let n_0 be the least integer such that $2^{an/2} > n^2 + n^{\log n + 1} + c_{T_1}$, for all $n \ge n_0$, where T_1 is the machine defined in Definition 2.6. We begin with Stage n_0. Prior to Stage n_0, let $A_{n_0 - 1} = A'_{n_0 - 1} = \emptyset$ and $D = \emptyset$ and $E = N$.

Stage n. (1) Let $j = \min(E)$. If $q_j(n) \le n^{\log n}$, then we let $D = D \cup \{j\}$ and $E = E - \{j\}$.

(2) Determine whether there exists an integer $i \in D$ having the property that $(\exists x, |x| = n)$ $[N_i^B(x)$ accepts for some B such that $A_{n-1} \subseteq B$ and $A'_{n-1} \subseteq \overline{B}]$. If yes, let i be the minimum integer having this property and fix x and an accepting computation path π of $N_i^B(x)$ and do the following.

(2.1) Add all queries "$z \in ?B$" in path π that are answered YES to A_{n-1} and add all queries "$z \in ?B$" in path π that are answered NO to A'_{n-1}.

(2.2) Search for a string y of length $an/2$ such that $y\{0,1\}^* \cap (A_{n-1} \cup A'_{n-1}) = \emptyset$. For each j, $1 \le j \le n$, add $y01^j$ to A_{n-1} if the jth bit of x is 1, or add it to A'_{n-1} otherwise. (Note that in stage k we add at most $k + k^{\log k}$ many strings to A. So $A_{n-1} \cup A'_{n-1}$ has size at most $n^2 + n^{\log n + 1} < 2^{an/2}$. Therefore, such a string y must exist.)

(2.3) Let $D = D - \{i\}$.

(3) If the search for i in step (2) fails, then do nothing.

(4) Let $A_n = A_{n-1}$ and $A'_n = A'_{n-1}$.

End of Stage i.

We let $A = \cup_{n=n_0}^{\infty} A_n$ and claim that A satisfies all requirements R_i.

First, assume that i is cancelled from D in some Stage n. Then we must have found, in Stage n, an x of length n such that (1) $N_i^A(x)$ accepts and (2) $\xi_A(n, y) = x$ for some y of length $an/2$ and hence $KT^A(x, t(n)) \le an/2 + c_{T_1} \le an$. So this x is a witness for requirement R_i.

Next, assume that i is never cancelled. We are going to show that $L(N_i, A)$ is finite. To see this, we first note that every i is eventually added to set D, because the runtime $q_i(n)$ is eventually dominated by the function $n^{\log n}$. So, we may assume that by some Stage n, $i \in D$, and all smaller indexes j, which are eventually to be cancelled, are already cancelled before Stage n. Let n_1 be the least integer such that the above holds. For each $n \ge n_1$, we observe that i is always considered but not cancelled in Stage n. Therefore, $N_i^B(x)$ rejects for all x of length n and for all sets B such that $A_{n-1} \subseteq B$ and $A'_{n-1} \subseteq \overline{B}$. In particular, $x \notin L(N_i, A)$ for all x of length $n \ge n_1$. This implies that $L(N_i, A)$ is a finite set. □

Remark. The above theorem actually holds for all sets $\overline{KT^A[n^r, t(n)]}$, $r > 0$.

THEOREM 5.3. *Let $0 < a < 1$ and t be a polynomial such that $t(n) \ge t_0(n)$. There exists an oracle A such that $\overline{KT^A[an, t(n)]}$ is not in P^A and is not P^A-immune.*

Proof. Define $L_A = \{x : \xi_A(|x| + 2, x) = 1x1\}$. Then it is obvious that $L_A \in P^A$. We are going to construct a set A such that $\overline{KT^A[an, t(n)]}$ is not in P^A and it contains L_A as an infinite subset. Let $e_{-1} = 0$ and $A_{-1} = \emptyset$.

At Stage $2i$, we consider the ith polynomial-time oracle machine M_i and make $KT^A[an, t(n)] \ne L(M_i, A)$. Select an integer $n = n_{2i} > e_{2i-1}$ that is sufficiently large. Find a string x of length n that is 1-random relative to A_{2i-1}. Simulate $M_i^{A_{2i-1}}(x)$. If the simulation accepts then do nothing and let $A_{2i} = A_{2i-1}$. Otherwise, if the simulation rejects, then find a string y of length $an/2$ so that no string in $y\{0,1\}^*$ has been queried in the computation of $M_i^{A_{2i-1}}(x)$. Let $m = n - |y|$. Let $A_{2i} = A_{2i-1} \cup \{y0^m01^j : 1 \le j \le n,$ the jth bit of x is $1\}$ so that $\xi_{A_{2i}}(n, y0^m) = x$. Finally, let $e_{2i} = \max\{q_i(n), t(n), 2n + 2\} + 2$.

At Stage $2i + 1$, we try to add one more string to L_A so that L_A has size greater than or equal to i. Choose $n = n_{2i+1} > e_{2i}$ such that $(1 - a)n > 3 \log n + c_{T_3} + 9$. (Recall the machine T_3 defined in §3.) We find a string x of length n that is 1-random relative to A_{2i}. Define

$$A_{2i+1} = A_{2i} \cup \{x01^j : j = 1 \text{ or } j = n+2 \text{ or } [2 \le j \le n+1 \text{ and the } (j-1)\text{th bit of } x \text{ is } 1]\}$$

so that $\xi_{A_{2i+1}}(n + 2, x) = 1x1$. Let $e_{2i+1} = t(n) + 1$.

We let $A = \cup_{i=0}^{\infty} A_i$. It is easy to verify that $KT^A[an, t(n)] \notin P^A$ and L_A is infinite. We claim that $L_A \subseteq \overline{KT^A[an, t(n)]}$ and complete the proof. To see this, we need to show that if $z \in L_A$, then $KT^A(z, t(|z|)) \ge a|z|$. Assume that $z \in L_A$. Then, $\xi_A(|z| + 2, z) = 1z1$ and so $\chi_A(z01) = 1$. We consider two cases.

First, if $z01$ is added to A in Stage $2i$ for some integer i, then z is of length n_{2i}. Note that in this case, we have made $e_{2i} \geq 2n_{2i} + 4$, and so $z01^{n_{2i}+2} \notin A$. That is, $\xi_A(n_{2i} + 2, z)$ ends with a zero. Thus, z cannot be in L_A.

Second, if $z01$ is added to A in Stage $2i + 1$ for some integer i, then it must be true that z is 1-random relative to A_{2i} and $|z| = n_{2i+1}$. Let $n = n_{2i+1}$. Using our standard argument in §§3 and 4, we see that for no program α of length an, can $U^{A_{2i}}(\alpha)$ query about any string of the form $z01^j$, $1 \leq j \leq n + 2$, in $t(n)$ moves. So, $KT^{A_{2i+1}}(z, t(n)) = KT^{A_{2i}}(z, t(n)) > an$. Next we observe that we have set $e_{2i+1} > t(n)$ so that all strings added to A later are of length longer than $t(n)$. This implies that $KT^A(z, t(n)) = KT^{A_{2i+1}}(z, t(n)) > an$ and so $z \in \overline{KT^A[an, t(n)]}$. □

The above results also hold for the set $LT^A[an, t(n)]$. The proofs are almost identical.

THEOREM 5.4. *Let $0 < a < 1$ and t be a polynomial such that $t(n) \geq r_0(n)$. There exists an oracle A such that $\overline{LT^A[an, t(n)]}$ is NP^A-immune.*

Proof. The proof is almost identical to Theorem 5.2. We leave it as an exercise. □

THEOREM 5.5. *Let $0 < a < 1$ and t be a polynomial such that $t(n) \geq r_0(n)$. There exists an oracle A such that $\overline{LT^A[an, t(n)]}$ is not in P^A and is not P^A-immune.*

Proof. The oracle is similar to the oracle of Theorem 5.3. Recall that in Theorem 4.3 we defined a machine $T_4^A(\langle n, \alpha \rangle)$ that simulates $V^A(\alpha, \sigma_j^n)$, $j = 1, \cdots, n$, and prints the outcomes in increasing order. We assumed that if $V^A(\alpha, \sigma_j^n)$ halts in $t(n)$ moves for all $j = 1, \cdots, n$, then $T_4^A(\langle n, \alpha \rangle)$ halts in $t_4(t(n))$ moves and so $U^A(\langle \tilde{T}_4, \langle n, \alpha \rangle \rangle)$ halts in $t'(n)$ moves. For every string w, let $x_w = \langle \vec{y}, \vec{z} \rangle$, where

$$\vec{y} = \{\sigma_j^{|w|} : 1 \leq j \leq |w|, \text{ the } j\text{th bit of } w \text{ is } 1\},$$

$$\vec{z} = \{\sigma_j^{|w|} : 1 \leq j \leq |w|, \text{ the } j\text{th bit of } w \text{ is } 0\}.$$

We construct set A to satisfy the following two conditions: (1) $\overline{LT^A[an, t(n)]} \neq L(M_i, A)$ for all $i \geq 0$, and (2) $\overline{LT^A[an, t(n)]}$ is not P^A-immune. For condition (1), our action in Stage $2i$ is almost identical to that of Theorem 5.3, except that our simulation of $M_i^{A_{2i-1}}$ is on input x_w for some 1-random string w (relative to A_{2i-1}), and that e_{2i} is chosen to be greater than $t'(n_{2i})$. This is sufficient because the KT^A-complexity of w is close to the LT^A-complexity of x_w.

For condition (2), we do the same as in Stage $2i+1$ of Theorem 5.3 with respect to size bound $(1 + a)n/2$ so that $L_A \subseteq \overline{KT^A[(1 + a)n/2, t(n)]}$, and let $e_{2i+1} > t'(n_{2i+1})$. Then, let $L'_A = \{x_w : w \in L_A\}$ and claim that $L'_A \subseteq \overline{LT^A[an, t(n)]}$. Let $x_w \in L'_A$ and so $w \in L_A$. Assume that $|w| = n_{2i+1}$ and $w01$ was added to A at Stage $2i + 1$. Then, our choice of $e_{2i+1} > t'(n_{2i+1})$ implies that $KT^A(w, t'(n_{2i+1})) \geq (1 + a)n/2$. Suppose otherwise that $x_w \in LT^A[an, t(n)]$. Then, the machine T_4 could print w in time $t_4(t(n))$, using a program α of length less than or equal to an, relative to A. This would imply that $KT^A(w, t'(n)) \leq an + 2\log n + c_4 + 2 < (1 + a)n/2$. This is a contradiction if n was chosen large enough. □

6. Open questions. A number of interesting questions about KT-complexity and LT-complexity remain open. We list a few of them here.

(1) Theorems 4.2 and 4.4 suggest that problems *MINKT* and *MINLT* may actually be provably complete for NP under some weak reducibility. It would be very interesting to either prove such an unrelativized result or to demonstrate that $MINKT^A$ is $\leq_T^{P,A}$-complete for NP^A for some oracle A. Vazirani and Vazirani [26] have shown

that a similar minimum Turing machine problem is indeed complete for NP under the $\leq_m^{RP}$-reducibility.

(2) Can the results like Theorems 3.1, 3.3, 4.1, 4.3 be improved to hold relative to random oracles?

(3) Can we prove that problem $MINKT^A$ does not have polynomial-size circuits relative to some oracle A? It is known that there exists an oracle A such that no NP^A-complete set has polynomial-size circuits [27]. However, the proof is a nonconstructive one and seems hard to apply to $MINKT^A$ because we do not know that $MINKT^A$ is complete for NP^A.

(4) What can we say about the space-bounded program-size complexity (or the KS-complexity)? Our preliminary study shows that the minimum-size KS-complexity problem has different structures from the problem $MINKT$. In particular, the principle (1) mentioned in §1 does not apply to strings with high KS-complexity.

Acknowledgments. The author thanks Paul Vitanyi for pointing out that Kolmogorov has considered the time-bounded version of program-size complexity in his original 1965 paper.

REFERENCES

[1] L. Adleman and K. Manders, *Reducibility, randomness and intractability*, in Proc. 9th ACM Symposium on Theory of Computing, 1977, pp. 151–163.

[2] D. Angluin, *Finding patterns common to a set of strings*, J. Comput. System Sci., 21 (1980), pp. 46–62.

[3] J. Balcázar and R. Book, *Sets with small generalized Kolmogorov complexity*, Acta Informatica, 23 (1986), pp. 679–688.

[4] J. Balcázar, J. Diaz, and J. Gabarró, *Structural Complexity* I, Springer-Verlag, Berlin, 1988.

[5] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, *Learnability and the Vapnik-Chervonenkis dimension*, J. Assoc. Comput. Mach., 36 (1989), pp. 273–282.

[6] M. J. Chung and B. Ravikumar, *Strong nondeterministic Turing reduction—a technique for proving intractability,* J. Comput. System Sci., 39 (1989), pp. 2–20.

[7] E. Gold, *Complexity of automaton identification from given data*, Inform. and Control, 37 (1978), pp. 302–320.

[8] J. Hartmanis, *Generalized Kolmogorov complexity and the structure of feasible computations*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, (1983), pp. 439–445.

[9] D. T. Huynh, *Resource-bounded Kolmogorov complexity of hard languages*, in Proc. Conference on Structure in Complexity Theory, Lecture Notes in Computer Science 223, 1986, pp. 184–195.

[10] M. Kearns, M. Li, L. Pitt, and L. Valiant, *On the learnability of boolean formulae*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 285–295.

[11] K. Ko, *On the notion of infinite pseudorandom sequences*, Theoret. Comput. Sci., 48 (1986), pp. 9–33.

[12] ———, *Separating the low and the high hierarchies by oracles*, Inform. and Comput., to appear.

[13] K. Ko, A. Marron, and W. Tzeng, *Learning string patterns and tree patterns from examples*, in Proc. 7th International Conference on Machine Learning, Morgan Kaufmann, 1990, pp. 384–391.

[14] K. Ko, P. Orponen, U. Schöning, and O. Watanabe, *What is a hard instance of a computational problem?*, in Proc. Conference on Structure in Complexity Theory, Lecture Notes in Computer Science 223, 1986, pp. 197–217.

[15] A. Kolmogorov, *Three approaches to the quantitative definition of information*, Prob. Inform. Trans., 1 (1965), pp. 1–7.

[16] L. Levin, *Randomness conservation inequalities; information and independence in mathematical theories*, Inform. and Control, 57 (1984), pp. 15–37.

[17] L. Li. and P. Vitanyi, *Two decades of applied Kolmogorov complexity*, in Proc. 3rd Conference on Structure in Complexity Theory, 1988, pp. 80–101.

[18] ———, *Inductive reasoning and Kolmogorov complexity*, in Proc. 4th Conference on Structure

in Complexity Theory, 1989, pp. 165–185.

[19] T. J. LONG, *Strong nondeterministic polynomial-time reducibility*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.

[20] L. LONGPRE, *Resource bounded Kolmogorov complexity, a link between computational complexity and information theory*, Ph.D. thesis, Cornell University, Ithaca, NY, 1986.

[21] L. PITT AND M. WARMUTH, *The minimum consistent DFA problem cannot be approximated within any polynomial*, in Proc. 20th ACM Symposium on Theory of Computing, 1989, pp. 421–432.

[22] U. SCHÖNING, *A low and a high hierarchy within* NP, J. Comput. System Sci., 27 (1983), pp. 14–28.

[23] M. SIPSER, *A complexity theoretic approach to randomness*, in Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 330–335.

[24] R. J. SOLOMONOV, *A formal theory of inductive inference, Part* 1, Inform. and Control, 7 (1964), pp. 1–22 and 224–254.

[25] L. VALIANT, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[26] U. V. VAZIRANI AND V. V. VAZIRANI, *A natural encoding scheme proved probabilistic polynomial complete*, Theoret. Comput. Sci., 24 (1983), pp. 291–300.

[27] C. WILSON, *Relativized circuit complexity*, J. Comput. System Sci., 31 (1985), pp. 169–181.

SIAM J. COMPUT.
Vol. 20, No. 5, pp. 987–997, October 1991

© 1991 Society for Industrial and Applied Mathematics
011

MINIMUM DIAMETER SPANNING TREES AND RELATED PROBLEMS*

JAN-MING HO†¶, D. T. LEE†, CHIA-HSIANG CHANG‡, AND C. K. WONG§

Abstract. The problem of finding a minimum diameter spanning tree (MDST) of a set of n points in the Euclidean space is considered. The diameter of a spanning tree is the maximum distance between any two points in the tree. A characterization of an MDST is given and a $\theta(n^3)$-time algorithm for solving the problem is presented. The authors also show that for a weighted undirected graph, the problem of determining if a spanning tree with total weight and diameter upper bounded, respectively, by two given parameters C and D exists is NP-complete. The geometric Steiner minimum diameter spanning tree problem, in which new points are allowed to be part of the spanning tree, is shown to be solvable in $O(n)$ time.

Key words. minimum diameter spanning tree, NP-complete problems, computational geometry, minimum enclosing circle, geometric Steiner trees

AMS(MOS) subject classifications. 68Q25, 68A20, 68R10

1. Introduction. The diameter of a weighted undirected graph $G = (V, E)$ is defined as the longest of the shortest paths among all the pairs of vertices V, where V is the set of vertices and E is the set of edges. The radius of G with respect to a specific vertex $v \in V$ is defined as the longest of the minimum paths emanating from v. A spanning tree of a graph $G = (V, E)$ is a connected graph $T = (V, E_T)$ without cycles.

Spanning tree related problems in a graph have been well studied [1], [13], [3]. So have the diameter problems in which the diameter is measured in terms of the number of edges, instead of the total weight [2]. What motivates this investigation is that we want to find a communication network among n nodes, where the communication delay is measured in terms of the total weight of a shortest path between them. A desirable communication network naturally is one that has a minimum diameter. We restrict the study of communication networks to the class of spanning trees. The *minimum diameter spanning tree* (MDST) problem is formally defined as follows.

PROBLEM 1 (MDST problem). *Given* a graph $G = (V, E)$ and a cost function $W(e) \in Z^+$ for all $e \in E$, *find* a spanning tree T for G such that

$$\max_{\textit{simple path } p \in T} \sum_{e \in p} W(e)$$

is minimized.

Let $\mathcal{D}_T$ denote the length of the diameter $D(T)$ of T. A tree T with minimum $\mathcal{D}_T$ is denoted T^*, and the length $\mathcal{D}_{T^*}$ is denoted D^* for short. Since we may have many spanning trees with the same minimum D^*, we would like, in this case, to find one with the minimum cost, i.e., the total weight of the edges in the tree T^* is

* Received by the editors June 13, 1989; accepted for publication (in revised form) December 17, 1990.

† Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208. The research of these authors was supported in part by National Science Foundation grants DCR 84-20814 and CCR 8901815.

‡ Department of Computer Science, Courant Institute of Mathematical Science, New York University, New York, New York 10012.

§ Thomas J. Watson Research Center, IBM Research Division, Yorktown Heights, New York 10598.

¶ Present address, Institute of Information Science, Academia Sinica, Taiwan, Republic of China.

minimum. We refer to this problem as the *minimum diameter minimum cost spanning tree* (MDMCST) problem.

PROBLEM 2 (MDMCST problem). *Given* a graph $G = (V, E)$ and a cost function $W(e) \in Z^+$ for all $e \in E$, *find* a spanning tree T for G such that $\mathcal{D}_T = \max_{simple\ path\ p \in T} \sum_{e \in p} W(e)$ is minimized and such that among all T's with the same $\mathcal{D}_{T^*} = \min_T \mathcal{D}_T$, $\sum_{e \in T^*} W(e)$ is minimized.

A similar problem known as the *bounded diameter spanning tree problem*, was shown to be NP-hard by Garey and Johnson [6]. In this problem, the measure of the diameter is in terms of the maximum *number* of edges in any path of the spanning tree. In a *centralized* communication network in which there is a designated *source* node, we may then define the *minimum radius spanning tree* (MRST) problem in a similar manner, except that we minimize the *radius* instead of *diameter*. That is, the maximum communication delay from the source node is to be minimized. The *minimum radius minimum cost spanning tree* (MRMCST) problem is similarly defined.

A variation of the MDST problem is that of the *Steiner* MDST, in which we connect a subset of nodes allowing additional nodes in the network. Specifically, we consider the following problem.

PROBLEM 3 (Steiner MDST problem). *Given* a graph $G = (V, E)$, a subset S of V, and a cost function $W(e) \in Z^+$ for all $e \in E$, *find* a Steiner tree $T = (V_T, E_T)$, where $S \subseteq V_T$, of G such that $\max_{simple\ path\ p \in T} \sum_{e \in p} W(e)$ is minimized.

In this paper we shall study the problem of finding a minimum diameter spanning tree (MDST) of a special graph, called *Euclidean graph*, induced by a set of n points in the Euclidean space, referred to as a *geometric* MDST problem. In other words, we are given a set S of n points in the Euclidean space, and are interested in finding a spanning tree connecting these points so that the length of its diameter is minimum. Also studied in the paper is the *geometric* Steiner MDST problem in which *Steiner* points, i.e., points not in S, are allowed so as to minimize the diameter of a spanning tree of S.

This paper is organized as follows. In §2 we give a characterization of a geometric minimum diameter spanning tree of n points and present a $\theta(n^3)$ algorithm for the geometric MDST problem. In §3 we show that the MDMCST and MRMCST problems are NP-hard by proving the corresponding *decision* versions of the problems NP-complete. In §4 we show that one Steiner point is sufficient for an optimal geometric Steiner MDST and outline a linear time algorithm for the problem. In §5, some open problems are given.

2. Geometric minimum diameter spanning tree. In this section, we first provide a proof for the existence of a geometric minimum diameter spanning tree (GMDST) of a set S of n points with a very simple topology, i.e., either *monopolar* or *dipolar*. A spanning tree of an n-point set S, $n \geq 3$, is said to be *monopolar* if there exists a point called the *monopole* such that all the remaining points are connected to it; and it is said to be *dipolar* if there exist two points called the *dipole* such that all the remaining points are directly connected to one of the two points in the dipole. The idea is to start with any optimal GMDST, and then to transform it into either a monopolar or a dipolar spanning tree without increasing the length of its diameter. In the following, we denote a path P that connects a sequence of points $A_0, A_1, \cdots, A_k$ as $P = (A_0, A_1, \cdots, A_k)$. Given a weighted undirected graph $G = (V, E)$, we use $\text{dist}_G(p, q)$ to denote the sum of the weights of the edges on a shortest path connecting vertices p and q, where $p, q \in V$. The following definition

will be used in proving the main theorem.

DEFINITION 1. An edge (A_{i-1}, A_i) is a center edge of a path $P = (A_0, A_1, \cdots, A_k)$ of points if $\max\{\text{dist}_P(A_0, A_{i-1}), \text{dist}_P(A_i, A_k)\}$ is minimized.

It is not difficult to show the following.

LEMMA 1. *Let (A_{i-1}, A_i) be a center edge of a path $P = (A_0, A_1, \cdots, A_k)$ of points. Then*

(1) $\text{dist}_P(A_0, A_{i-1}) \leq \text{dist}_P(A_{i-1}, A_k)$ *and*

(2) $\text{dist}_P(A_i, A_k) \leq \text{dist}_P(A_0, A_i)$.

LEMMA 2. *There exists a* GMDST *of a set S of n points which is either monopolar ($n \geq 3$) or dipolar ($n \geq 4$).*

Proof. Consider an arbitrary GMDST T of S. If there exists a diameter of T, which has only two edges, i.e., $D(T) = (A_0, A_1, A_2)$, then a monopolar spanning tree T' can be constructed by connecting every point in T directly to A_1. Let $|P, Q|$ denote the Euclidean distance between points P and Q. For any pair of points P and Q on T', we have

$$\begin{aligned} \text{dist}_{T'}(P, Q) &= |P, A_1| + |A_1, Q| \\ &\leq \text{dist}_T(P, A_1) + \text{dist}_T(A_1, Q) \\ &\leq |A_0, A_1| + |A_1, A_2| \\ &= \mathcal{D}_T. \end{aligned}$$

The first inequality follows from the triangle inequality and the second from the definition of diameter. Note that equality holds when $\{P, Q\} = \{A_0, A_2\}$. Therefore $\mathcal{D}_{T'} = \mathcal{D}_T$.

Assume that every diameter of T contains more than two edges. Let $D(T) = (A_0, A_1, \cdots, A_k)$, $k \geq 3$ be an arbitrary diameter of T, and the center edge of $D(T)$ be denoted as $(A_{i-1}, A_i), 1 \leq i \leq k$. By deleting the edge (A_{i-1}, A_i) from T, we obtain two subtrees T_{i-1} and T_i with T_{i-1} containing A_{i-1} and T_i containing A_i. A dipolar spanning tree T'' can be constructed by connecting all the vertices in T_{i-1} directly to A_{i-1} and vertices in T_i directly to A_i. Consider the distance between any two points P and Q on T''. If P and Q belong to different subtrees, then their distance is obviously less than $\mathcal{D}_T$. Otherwise, without loss of generality, assume both points are contained in the subtree T_i. Note that (A_{i-1}, A_i) is the center edge of $(A_0, A_1, \cdots, A_k)$. From Lemma 1 we have $\text{dist}_T(A_i, A_k) \leq \text{dist}_T(A_0, A_i)$. We have

$$\begin{aligned} \text{dist}_{T''}(P, Q) &= |P, A_i| + |A_i, Q| \\ &\leq \text{dist}_T(P, A_i) + \text{dist}_T(A_i, Q) \\ &\leq \text{dist}_T(A_i, A_k) + \text{dist}_T(A_i, A_k) \\ &\leq \text{dist}_T(A_0, A_i) + \text{dist}_T(A_i, A_k) \\ &= \mathcal{D}_T. \end{aligned}$$

That is, $\text{dist}_{T''}(P, Q)$ is always no greater than $\mathcal{D}_T$, and since T is a GMDST, $\mathcal{D}_{T'} = \mathcal{D}_T$. This completes the proof. □

Based on Lemma 2 we can design Algorithm find_min_of_MPST and Algorithm find_min_of_DPST that enumerate all the monopolar and dipolar spanning trees, denoted MPST and DPST, respectively, and find the optimal one.

Detailed discussions of the two algorithms are presented in the following subsections.

2.1. Algorithm find_min_of_MPST. For each point $p \in S$, we construct a monopolar spanning tree by connecting each point in the set $S - \{p\}$ to p. The diameter for the MPST centered at p is the sum of the distance between p and its farthest neighbor and the distance between p and its second farthest neighbor. The Algorithm find_min_of_MPST(S) returns the MPST, centered at a point p^*, whose diameter is the minimum. This can be done in $O(n \log n)$ time using the farthest neighbor Voronoi diagram (FNVOD) [9], [14].

In the algorithm, we first construct the second-order FNVOD of S [10], [14], in which each farther neighbor Voronoi region is characterized by a pair of points such that any point in the region is farther from these two points than from any other points in S. For each point $p \in S$, the region containing p can be obtained from any point-location algorithm given in [5], [8], [12], and the sum of the distances from p to the two points associated with the region, denoted $d(p)$, is calculated. The point p^* minimizing $d(p)$ is then the monopole of the MPST.

The second-order FNVOD algorithm is similar to the kth-order nearest neighbor Voronoi diagram algorithm [10], [14], and can be computed in $O(n \log n)$ time as follows.

1. Compute the two outermost convex layers of the set S of n points in $O(n \log n)$ time, where the outermost convex layer is exactly the convex hull CH1 of S and the second outermost convex layer is the convex hull CH2 of the set of points in S with points in CH1 removed.
2. Calculate the clockwise supporting line from each point on CH1 to CH2 in $O(n)$ time. Similarly calculate the counterclockwise supporting line from each point on CH1 to CH2. Using the supporting line information thus calculated, the convex hull of S can be updated in $O(1)$ time when a point on the convex hull CH1 is deleted.
3. The FNVOD of the set S is constructed from the convex hull of S in $O(n \log n)$ time [9], [14].
4. To obtain the second-order FNVOD, the region associated with point p on CH1 is further subdivided by considering its neighbors on the FNVOD and the points on CH2 that become vertices of the new convex hull when p is deleted from the set S (cf. [10]). The set of new points, if any, can be obtained from the information calculated in step 2. Using an algorithm similar to that proposed by Gowda, Kirkpatrick, Lee, and Naamad [7], we can construct the second-order FNVOD in $O(n \log n)$ time, as shown by Lemma 3 in the following.

Let S be a set of n points; $p_1, \cdots, p_{h1}$ be points on CH1; and $q_1, \cdots, q_{h2}$ on CH2, both sequences specified in counterclockwise direction. Note that when a point $p_i \in$ CH1 is deleted from CH1, a *subchain* PC_i of points $c_1^i, \cdots, c_{k_i}^i$ of CH2, which may be empty, appears as new points on CH1. The chain is called the *patch-up chain* associated with a point $p_i \in$ CH1. Denote the number of regions in the FNVOD adjacent to the region associated with a point p_i on the convex hull as s_i. Then we have the following lemma.

LEMMA 3.

(1)

$$\sum_{i=1}^{h1} s_i = O(n);$$

(2)

$$\sum_{i=1}^{h1} k_i = O(n).$$

Proof. The first equality follows from the fact that there are $O(n)$ edges in the FNVOD. The second equality follows from the assertion that a point $q_j \in$ CH2 may appear on at most two patch-up chains. This can be shown as follows. Let p_{i-1}, p_i, p_{i+1} be three consecutive points on $CH1$, where the arithmetics are taken modulo $h1$. Let Δ_i denote the triangle $\Delta p_{i-1}p_ip_{i+1}$. If q_j appears on the patch-up chain PC_i, then q_j must be interior to Δ_i. By convexity, the two triangles Δ_i and Δ_k intersect at their interior if and only if i and k are adjacent. In other words, q_j can also appear in the interior of either Δ_{i-1} or Δ_{i+1}, but not both. This completes the proof. □

2.2. Algorithm find_min_of_DPST. Given a dipolar spanning tree with dipole p_i and p_j, the sets of vertices which are connected to p_i and p_j are denoted as V_i and V_j, respectively. In searching for an optimal DPST, as we show below, we may restrict ourselves to only those dipolar spanning trees such that there exists a point $q \in V_i$ and $|q, p_j| > |q', p_j|$, for all $q' \in V_j$, and vice versa. This condition is called the *stability condition.*

LEMMA 4. *Let T be a dipolar spanning tree with dipole p_i and p_j. If T does not satisfy the stability condition, then there exists a monopolar spanning tree whose diameter is no greater than that of T.*

Proof. Without loss of generality, assume $|q, p_j| \leq |q^*, p_j|$ for all $q \in V_i$, where $|q^*, p_j| = \max_{q' \in V_j} |q', p_j|$. Then the diameter of the monopolar spanning tree centered at p_j is no greater than that of T. □

The Algorithm find_min_of_DPST(S) can be implemented in $O(n^3)$ time as described in the following. Basically, we consider all pairs of points p_i and p_j of the set S as possible dipoles of the GMDST, and select the pair that gives a minimum diameter. Consider now any pair p_i and p_j, $i \neq j$. We now want to find two circles K_i and K_j centered at p_i and p_j with radii $R_i = |p_i, e_i|$ and $R_j = |p_j, e_j|$, respectively, covering the set of points such that the sum of R_i, R_j and $|p_i, p_j|$ is a minimum, where $e_i, e_j \in S$ are on circles K_i and K_j, respectively. We proceed as follows. Points other than p_i and p_j are sorted, according to their distances with respect to p_i, into a nondecreasing list $L_{i,j}$. The kth point in the list $L_{i,j}$ is designated as $L_{i,j}[k]$. By traversing the list, we can identify all the possible four-point combinations (e_i, p_i, p_j, e_j), each of which determines the diameter of a dipolar spanning tree, and find the best dipolar diameter with dipole p_i and p_j. At the kth step of traversing, we consider the tree $T(i, j, k)$ in which every point $L_{i,j}[l], 1 \leq l \leq k \leq n-2$, is directly connected to p_i, and all others are connected to p_j. The length of its diameter is determined by the following function $f(i, j, k)$, given the stability condition

$$f(i,j,k) = |L_{i,j}[k], p_i| + \max_{l=k+1}^{n-2} |L_{i,j}[l], p_j| + |p_i, p_j|. \tag{1}$$

Note that $f(i, j, k)$ can be updated in $O(1)$ time if $\max_{l=k+1}^{n-2} |L_{i,j}[l], p_j|$ is given, and testing for the stability condition also takes $O(1)$ time. The minimizer k_{ij} for $f(i, j, k_{ij})$ can be found in $O(n)$ time. By exhaustively searching for all possible pairs p_i and p_j that minimize $f(i, j, k_{ij})$, we can obtain an optimal DPST in $\theta(n^3)$ time. We thus conclude with the following theorem. Detailed implementation of the Algorithm find_min_of_DPST is given in Fig. 1.

```
d := ∞;
for each point p_i ∈ S do
    construct a sorted array L' of the set S − {p_i} of points in nondecreasing
    order of their distances to the point p_i;
    for each point p_j ∈ S − {p_i} do
        L := the sorted array constructed by deleting p_j from L';
        Note: L[k] denotes the kth element in the array L.
        m_2 = i_2 = n − 2;          r_2 := |L[i_2], p_j|;
        m_1 = i_1 = n − 3;          r_1 := |L[i_1], p_i|;
        s = r_1 + r_2;
        k := n − 3;
        while k ≥ 1 do  {
            if |L[k], p_j| > r_2 then  {
                i_1 := k;               r_1 := |L[i_1], p_i|;
                if r_1 + r_2 < s then
                    s := r_1 + r_2;
                    m_1 := i_1;         m_2 := i_2;
                    i_2 := i_1;         r_2 := |L[i_2], p_j|;
            }
            k := k − 1;
        }
        if s + |p_i, p_j| < d then  {
            d := s + |p_i − p_j|;
            c_1 := p_i;             c_2 := p_j;
            e_1 := L[m_1];          e_2 := L[m_2];
        }
    end; { of for each p_j }
end. { of for each p_i }
```

FIG. 1. *Description of Algorithm find_min_of_DPST.*

THEOREM 1. *Given a set S of n points, the geometric minimum diameter spanning tree for S can be found in $\theta(n^3)$ time and $O(n)$ space.*

3. NP-completeness of the BDBCST problem. In this section, we will show a *decision* version, called *bounded diameter bounded cost spanning tree* (BDBCST) problem, of the optimization MDMCST problem, for general graphs to be NP-complete.

PROBLEM 4 (BDBCST problem). *Given* a graph $G = (V, E)$, a cost function $W(e) \in Z^+$ for all $e \in E$, and positive integers C and D, *is there* a spanning tree T for G such that $\sum_{e \in T} W(e) \leq C$ and $\sum_{e \in p} W(e) \leq D$ for all simple paths p in T?

THEOREM 2. BDBCST *is* NP*-complete.*

Proof. BDBCST is obviously in NP.

Let $\mathcal{C} = \{C_1, C_2, \cdots, C_q\}$ be an instance of 3SAT over variable set $\{X_1, X_2, \cdots, X_n\}$. We will construct a graph $G = (V, E)$ and weight function W such that $\mathcal{C}$ is satisfiable if and only if there is a spanning tree T for G such that $\sum_{e \in T} W(e) \leq 3n + 3q + 5$ and $\sum_{e \in p} W(e) \leq 10$ for all simple paths p in tree T.

The construction of G is as follows (Fig. 2). G contains the following vertices: the *true setting node* t, the *variable nodes* $\{x_1, \bar{x}_1, x_2, \bar{x}_2, \cdots, x_n, \bar{x}_n\}$, the *clause nodes* $\{c_1, c_2, \cdots, c_q\}$, and a special node s. G contains the following edges: the *assignment edges* $\{(t, x) |$ for all variable node $x\}$, the *consistency edges* $\{(x_i, \bar{x}_i) | i = 1$ *to* $n\}$, the *containment edges* $\{(u_i, c_j) \mid u_i = x_i$ or $\overline{x_i}$ depending on whether $C_j \in \mathcal{C}$ contains literal X_i or $\overline{X_i}$, respectively$\}$, and edge (t, s). The assignment edges have weight 2; the consistency edges have weight 1; the containment edges have weight 3; and edge

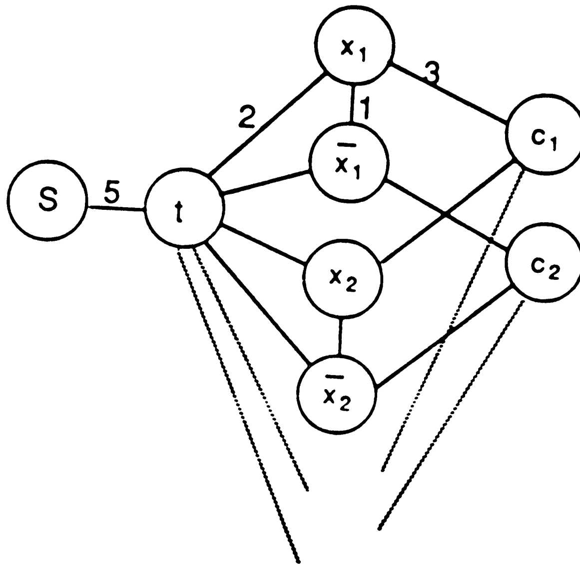


FIG. 2. *Construction of G in proof of Theorem* 2.

(t, s) has weight 5. It is clear that this is a polynomial-time construction.

Suppose $\mathcal{C}$ is satisfiable. There is an assignment such that every clause in $\mathcal{C}$ contains at least one true variable. We may have a spanning tree T consisting of the following edges: edge (t, s), (t, x_i) for all i where variable X_i is assigned true or $(t, \bar{x}_i)$; otherwise, all the consistency edges, and the containment edges (x_{i_j}, c_i) for all i where X_{i_j} is the first true variable in clause C_i. It is easy to see that $\sum_{e \in T} W(e) = 3n+3q+5$. Since the weight of the path from a leaf node to node t is at most 5, there is no path in T with weight greater than 10.

Suppose there is a spanning tree T for G such that $\sum_{e \in T} W(e) \leq 3n + 3q + 5$ and no simple path p in T with weight greater than 10. Since there are $2n + q + 2$ vertices in G, we have $2n + q + 1$ edges in T. T must include the edge (t, s), since (t, s) is the only edge incident to vertex s. For clause nodes are connected only by the containment edges, there are at least q containment edges in T. Assume the spanning tree T consists of i assignment edges, j consistency edges, $k + q$ containment edges, and the edge (t, s). We have the following inequalities:

$$i, j, k \geq 0, \tag{2}$$

$$j \leq n, \tag{3}$$

$$i + j + k = 2n, \tag{4}$$

$$2i + j + 3k \leq 3n. \tag{5}$$

Assume $j < n$. From $(5) - 2 \times (4)$, we have $k \leq j - n < 0$, a contradiction. Therefore, T must include all n consistency edges. Substituting $j = n$ into the above formulas, we have

$$i + k = n, \tag{6}$$

$$2i + 3k \leq 2n. \tag{7}$$

From $(7) - 2 \times (6)$, we have $k \leq 0$. Therefore, T contains exactly q containment edges, n consistency edges, and n assignment edges.

We claim that all the clause nodes are adjacent only to those variable nodes which are adjacent to node t in spanning tree T. Suppose this claim is not true. Because clause nodes are only adjacent to variable nodes in graph G, there must exist a variable node x and a clause node c such that x is adjacent to c but x is not adjacent to node t in T (see Fig. 2). The path from c to t which must go through x is of weight 6. So, the path from s to c in T is of weight 11, which is a contradiction.

Since T contains all the consistency edges, T contains exactly one of the edges (t, x_i) and $(t, \bar{x}_i)$ for all i. Assigning "true" to all the variables adjacent to t and "false" to their complements, all the clauses in E are satisfied for every clause containing at least one true variable. Hence, 3SAT is polynomially reducible to BDBCST. □

Note that the cost constraint $3n + 3q + 5$ is exactly the minimum cost C^* of T and the diameter constraint 10 is the minimum possible diameter of the graph G used in the proof. The above proof actually reveals that even in the class of spanning trees with minimum diameter, finding one with minimum cost (MDMCST problem) is NP-hard. It also shows that in the class of spanning trees with minimum total cost, the problem (the MCMDST problem) of finding a spanning tree with minimum diameter is NP-hard. Thus, the following corollary is self-evident.

COROLLARY 1. (1) *The* MDMCST *and the* BDMCST *problems are* NP*-hard.*

(2) *The* MCMDST *and the* BCMDST *problems are* NP*-hard.*

The same arguments used in the proof of Theorem 2 can be used to prove the NP-completeness of the BRBCST problem except that the notion of diameter is replaced by the notion of radius, that the diameter bound 10 is replaced by the radius bound 5, and that the vertex t is taken as the distinguished vertex. Similar conclusions can be drawn for the optimization problems.

THEOREM 3. *The* BRBCST *problem is* NP*-complete.*

COROLLARY 2. (1) *The* MRMCST *and the* BRMCST *problems are* NP*-hard.*

(2) *The* MCMRST *and the* BCMRST *problems are* NP*-hard.*

4. Geometric Steiner minimum diameter spanning tree. In this section, we consider a geometric Steiner tree problem for the set S of n input points. A *geometric Steiner minimum diameter spanning tree* (GSMDST) is a tree T_S that includes all the points of S and some vertices on the tree T_S not in S, and the diameter of the tree is minimized.

We first give a characterization of an optimal GSMDST and present an efficient algorithm to compute one.

LEMMA 5. *There exists an optimal* GSMDST T_S *for* S *which is monopolar.*

Proof. Consider an arbitrary GSMDST T_S of S. According to Lemma 2, T_S is either monopolar or dipolar. In the former case we are done. Let us assume points c_1 and c_2, where $c_1 \neq c_2$, are the dipole of T_S.

Let the set of points, excluding $\{c_2\}$, that are connected to c_1 be denoted V_1. The set V_2 is similarly defined. Among points in the set V_i, let e_i^1, $i = 1, 2$, be the farthest from the center c_i, and e_i^2 be the second farthest. Let $d_i^j, i, j = 1, 2$, denote the distance $|e_i^j, c_i|$; and α be the distance $|c_1, c_2|$. Assume that the straight line $\overleftrightarrow{c_1, c_2}$ passing through c_1 and c_2 is positioned horizontally. Note that neither e_1^1 nor e_2^1 lies on $\overleftrightarrow{c_1, c_2}$. Otherwise, T_S can be transformed into a monopolar tree $T_S' = (V_S, E_S')$ centered at c_1 or c_2 without increasing the length of the diameter. Furthermore, the vertical projection of e_1^1 (respectively, e_2^1) on $\overleftrightarrow{c_1, c_2}$ does not lie on the segment $\overline{c_1, c_2}$, otherwise c_1 (respectively, c_2) can be *moved* toward the other center and the length of the diameter can be shortened.

We will show that

$$d_i^1 + d_i^2 = d_1^1 + d_2^1 + \alpha, \tag{8}$$

where $i = 1, 2$, by the following arguments:

(1)

$$d_i^1 + d_i^2 \leq d_1^1 + d_2^1 + \alpha = \mathcal{D}_{\mathcal{T}_S}.$$

(2) If

$$d_1^1 + d_1^2 < d_1^1 + d_2^1 + \alpha,$$

then there exists an ϵ-neighborhood $N_\epsilon(c_1)$ of the point c_1 such that $\exists c_1' \neq c_1, c_1' \in N_\epsilon(c_1) \cap \overline{c_1 c_2}$, and

$$d_1'^1 + d_1'^2 < d_1^1 + d_2^1 + \alpha;$$

where $d_1'^1$ is the longest distance among those from points in the set V_1 to the point c_1', and $d_1'^2$ is the second farthest. Since c_1, c_2, and e_1^1 are not collinear, by the triangle inequality it is easy to show that

$$d_1'^1 + d_2^1 + \alpha' < d_1^1 + d_2^1 + \alpha.$$

In other words, we can construct a new Steiner tree T_S' with diameter $D(T_S') = (e_1'^1, c_1', c_2, e_2^1)$, whose length is shorter than $\mathcal{D}_{\mathcal{T}_S}$. This is a contradiction. Similarly,

$$d_2^1 + d_2^2 < d_1^1 + d_2^1 + \alpha$$

does not hold.

By (8), we have $d_1^2 = d_2^1 + \alpha$ and $d_2^2 = d_1^1 + \alpha$. By definition, we have $d_2^1 \geq d_2^2$ and $d_1^1 \geq d_1^2$, and

$$d_2^1 \geq d_2^2 = d_1^1 + \alpha \geq d_1^2 + \alpha = d_2^1 + 2\alpha.$$

Thus $\alpha = 0$. This contradicts the assumption that c_1 and c_2 are distinct. □

Lemma 5 implies that at most one Steiner point is needed for the construction of a monopolar geometric Steiner minimum diameter spanning tree.

In the next lemma, we establish the relationship between the monopole of the Steiner MDST and the center of the smallest enclosing circle.

LEMMA 6. *Let c_M be the center of the smallest enclosing circle of S. The monopolar Steiner tree T_M with c_M as the monopole has a minimum diameter.*

Proof. Let $f_S(x)$ denote the length of the diameter of a monopolar Steiner tree T_S with x being the monopole. Suppose there exists a monopolar Steiner tree with c being the monopole such that $c \neq c_M$ and $f_S(c) < f_S(c_M)$. Let $e_1, e_2 \in S$ be the farthest and second farthest points from c.

Case 1. $|e_1, c| = |e_2, c|$. In this case the diameter $f_S(c) = 2|e_1, c|$, which is equal to the diameter of the circle C centered at c and enclosing S. Since $f_S(c_M)$ is exactly the diameter of the smallest enclosing circle of S, it is impossible that $f_S(c) < f_S(c_M)$.

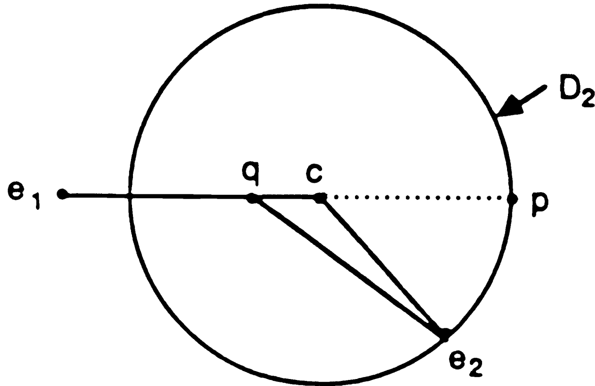


FIG. 3. *Illustration for the proof of Lemma* 6.

Case 2. $|e_1, c| > |e_2, c|$. Let D_1 and D_2 be the two concentric circles centered at c such that e_1 and e_2 lie, respectively, on D_1 and D_2. Denote p as the point of intersection of D_2 and the line $\overleftrightarrow{ce_1}$ such that c is between p and e_1. Suppose $p \in S$. Consider the circle C' centered at the midpoint c' of $\overline{pe_1}$ with radius $|p, e_1|/2$. It is obvious that C' encloses S and that $f_S(c') = f_S(c)$ is equal to the diameter of C'. Hence it is impossible that $f_S(c) < f_S(c_M)$. Now let us assume that $p \notin S$. Consider a point q on line segment $\overline{ce_1}$ in the neighborhood $N_\epsilon(c)$ of c such that e_1 and e_2 remain the farthest and second farthest points of q. We have $f_S(q) = |q, e_2| + |q, e_1|$. By the triangle inequality, we have $f_S(c) > f_S(q)$ (see Fig. 3). Thus c cannot be the monopole of a GSMDST, a contradiction. □

We conclude with the following theorem.

THEOREM 4. *The geometric Steiner minimum diameter spanning tree problem is reducible to the minimum enclosing circle problem, and hence can be solved in $O(n)$ time.*

Proof. We simply use the 1-center algorithm proposed by Dyer [4] or Megiddo [11], and use the center as the monopole of the monopolar Steiner tree. □

5. Conclusion. We have considered a new class of problems pertaining to the diameter of spanning trees. We have presented a $\theta(n^3)$-time algorithm for finding a minimum diameter spanning tree of a set of n points in the plane. The result actually is applicable to any complete graph whose edge weight satisfies the triangle inequality. We have considered the Steiner minimum diameter spanning tree problem and presented a linear-time algorithm. Furthermore, we have shown that the problem of finding a minimum diameter minimum cost spanning tree of a general graph is NP-hard. We conjecture that the problem of deciding if a spanning tree of a set of n points in the plane (so that the total cost and diameter are both bounded) exists is also NP-complete. Whether or not the $O(n^3)$ time bound for finding a minimum diameter spanning tree can be improved is of great interest.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] W. G. BROWN, *Reviews in Graph Theory*, American Mathematical Society, Providence, RI, 1980.

[3] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 724–742.

[4] M. E. Dyer, *Linear time algorithms for two- and three-variable linear programs*, SIAM J. Comput., 13 (1984), pp. 31–45.

[5] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[6] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of* NP-*completeness*, W. H. Freeman, San Francisco, CA, 1979.

[7] I. G. Gowda, D. G. Kirkpatrick, D. T. Lee, and A. Naamad, *Dynamic Voronoi diagrams*, IEEE Trans. Inform. Theory, IT-29 (1983), pp. 724–731.

[8] D. G. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[9] D. T. Lee *Farthest neighbor Voronoi diagrams and applications*, Tech. Report #80-11-FC-04, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, November 1980.

[10] ———, *On finding k-nearest neighbor Voronoi diagrams in the plane*, IEEE Trans. Comput., C-31 (1982), pp. 478–487.

[11] N. Megiddo, *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput., 12 (1983), pp. 759–776.

[12] F. P. Preparata, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473–482.

[13] R. C. Prim, *Shortest connecting networks and some generalizations*, Bell System Tech. J., 36 (1957), pp. 1389–1401.

[14] M. I. Shamos and D. Hoey, *Closest-point problems*, in 16th Annual IEEE Symposium on Foundations of Computer Science, October 1975, pp. 151–162.

SIAM J. COMPUT.
Vol. 20, No. 5, p. 998, October 1991
© 1991 Society for Industrial and Applied Mathematics
012

ERRATUM: FACTORING POLYNOMIALS OVER ALGEBRAIC NUMBER FIELDS*

SUSAN LANDAU†

The proof of Corollary 2, which was originally presented in [1], is false. A correct proof of the theorem (by the fact that intersections of fields can be computed in polynomial time) appears in [2]. However, an even simpler proof was pointed out by Hendrik Lenstra, namely, that the problem of field intersection reduces to the intersection of subspaces of a vector space. That can be done by linear algebra.

Acknowledgments. My thanks to Hendrik Lenstra for kindly pointing out the error.

REFERENCES

[1] S. LANDAU, *Factoring polynomials over algebraic number fields*, SIAM J. Comput., 14(1985), pp. 184–195.

[2] S. LANDAU AND G. MILLER, *Solvability by radicals is in polynomial time*, J. Comput. System Sci., 30(1985), pp. 179–208.

* Received by the editors May 22, 1991; accepted for publication May 22, 1991.

† Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003.

SIAM J. COMPUT.
Vol 20, No. 6, pp. 999-1007, December 1991
© 1991 Society for Industrial and Applied Mathematics
001

CREW PRAMs AND DECISION TREES*

NOAM NISAN†

Abstract. This paper gives a full characterization of the time needed to compute a boolean function on a CREW PRAM with an unlimited number of processors.

The characterization is given in terms of a new complexity measure of boolean functions: the "block sensitivity," a generalization of the well-known "critical sensitivity" measure. The block sensitivity is also shown to relate to the boolean decision tree complexity, and the implication is that the decision tree complexity also fully characterizes the CREW PRAM complexity. This solves an open problem of Wegener.

The results imply that changes in the instruction set of the processors or in the capacity of the shared memory cells do not change by more than a constant factor the time required by a CREW PRAM to compute any boolean function. Moreover, it is shown that a seemingly weaker version of a CREW PRAM, the CROW PRAM, can compute functions as quickly as a general CREW PRAM. This solves an open problem of Dymond and Ruzzo.

Finally, the results have implications regarding the power of randomization in the boolean decision tree model. It is shown that in this model, randomization may achieve only a polynomial speedup over deterministic computation. This was known for Las Vegas randomized computation; it is also proven for one-sided error computation (a quadratic bound) and two-sided error (a cubic bound).

Key words. PRAM, decision trees, block sensitivity

AMS(MOS) subject classifications. 68Q05, 68Q10

1. Introduction.

1.1. CREW PRAMs. The PRAM (Parallel Random Access Machine) is the "standard" model for parallel computation. In the PRAM model the processors communicate via shared memory cells. We will be interested in the inherent limitations of this model that are due to its basic communication mechanism, and will thus consider the "ideal" PRAM, a model that has no other constraints.

An *ideal* PRAM consists of an unbounded number of processors and an unbounded number of common memory cells that can be read and written by any processor. Each processor has its own local memory and possibly unlimited computational power. A PRAM computes a function in the following manner: The input is placed in the common memory cells, and then the computation proceeds in cycles. At each cycle each processor may read one memory location, do any computation using the information it knows, and write any information into one memory cell. Several variants of the PRAM model have been defined, which differ from each other in the way they handle memory access conflicts. Perhaps the most natural variant, and the one we will be considering, is the CREW PRAM. For a CREW (Concurrent Read Exclusive Write) PRAM several processors may read from the same location at the same time, but two or more processors may *never* attempt writing into the same location at the same time.

A key result bounding the power of ideal CREW PRAMs is the following theorem by Cook, Dwork, and Reischuk [CDR]: Computing the OR function on n variables by a CREW PRAM takes $\Omega(\log n)$ parallel time. This result is tight since any function can be computed in $\log n$ time on this model. Actually, [CDR] proved a more general result as they gave a lower bound on the time needed to compute a function on a CREW PRAM in terms of the function's "critical sensitivity."

* Received by the editors August 28, 1989; accepted for publication (in revised form) November 30, 1990. This work was done while the author was a student at the University of California at Berkeley and was supported by a grant from Digital Equipment Corporation.

† Department of Computer Science, Hebrew University of Jerusalem, Jerusalem 91904, Israel.

We consider a generalization of the "critical sensitivity" measure: the "block sensitivity." We show that the [CDR] lower bound can be extended to block sensitivity, and, moreover, that the block sensitivity fully characterizes the complexity on the ideal CREW PRAM model. We achieve this result by further relating the block sensitivity to the decision tree complexity, and thus we alternatively characterize the CREW complexity in terms of the decision tree complexity.

For a boolean function f, let $CREW(f)$ denote the CREW PRAM complexity of f (i.e., the time needed to compute f on an ideal CREW PRAM with an unlimited number of processors and memory cells), let $bs(f)$ denote the block sensitivity of f, and let $D(f)$ denote the boolean decision tree complexity of f.

THEOREM 1.

$$CREW(f) = \Theta(\log bs(f)) = \Theta(\log D(f)).$$

Moreover, the lower bound holds for an ideal PRAM with unlimited computational power for each processor and unlimited capacity of the common memory cells, while the upper bound requires only reasonable power for each processor and 1-bit memory cells. As a corollary, we get that the computational power of a single processor, and the capacity of the common memory cells, does not make a difference in this model (as long as we do not limit the number of processors, and ignore uniformity questions).

These results also apply to a weaker model than the CREW PRAM: the CROW PRAM (Concurrent Read *Owner* Write) introduced by Dymond and Ruzzo [DR]. For a CROW PRAM, each memory cell is preassigned to some processor who is said to "own" the memory cell. The only processor who may write into a memory cell is the owner, although all processors may read it. This model is clearly a special case of the CREW model, but we show that the parallel time needed to compute a function on this model is equal (up to a constant factor) to the time needed by a CREW PRAM. Let $CROW(f)$ be the parallel time needed to compute f by an ideal CROW PRAM.

THEOREM 2.

$$CROW(f) = \Theta(CREW(f)).$$

This result is particularly surprising as it is not achieved by simulation, and only applies to functions on a *full* domain. We actually show that a simulation result is impossible by giving a function on a *partial domain* that separates these two models by a factor of $\log n$.

1.2. Boolean decision trees. The boolean decision tree model is perhaps the simplest computational model for general boolean functions. A deterministic boolean decision tree computes a boolean function by repeatedly reading input bits until the function can be determined from the bits accessed. The decision of which bit to read at any time may depend on the previous bits read, and is determined by them. The only cost associated with this computation is the number of bits read; all other computation is free. The cost of an algorithm is the number of bits read on the worst case input, and the deterministic complexity of a function is the cost of the best deterministic algorithm for this function.

A randomized decision tree algorithm is also allowed to flip coins in order to determine the next input bit to be accessed. The cost of an algorithm is the expected number of locations examined on the worst case input. The complexity of a function is the cost of the best algorithm for this function. We distinguish between three kinds of probabilistic algorithms: zero error, one-sided error, and two-sided error. In zero-error computation of f, the algorithm must always be correct—no errors are allowed.

For one-sided error computation the algorithm must always reject any string not in the language, and must accept any string in the language with probability of at least $\frac{1}{2}$. (Here we identify the function f with the language $\{x|f(x)=1\}$.) A two-sided error algorithm may err in both cases, but must give the correct answer with probability of at least $\frac{3}{4}$.

This model has been studied extensively in several contexts. The complexity of *graph* properties in this model has been investigated [Ro], [RV], [KK], [KSS]. The relation to Oracle Turing machines has been pointed out several times [BI], [Ta], [IN]. It is also related to sublinear time Turing machine computations [IN]. The randomized complexity in this model has also been studied [Y1], [Y2], [Sn], [K], [SW].

In this paper we deal with the power of randomization in the boolean decision tree model. Snir, [Sn] (see also [SW]), first showed that randomization "helps": he exhibited a function with deterministic complexity n, but Las Vegas (zero error) randomized complexity of only $O(n^{0.753\cdots})$. Saks and Wigderson [SW] conjecture that this is the optimal speedup possible by randomization, and were able to prove it for a particular subclass of functions; for general functions the conjecture is still open. A more general result by Blum implies some bounds on the speedup possible by *zero-error* randomization: zero-error randomized decision tree algorithms can give at most a quadratic speedup. In this paper we show that even allowing error, randomization may only give a polynomial speedup.

Let $R_1(f)$ and $R_2(f)$ denote the one-sided error and two-sided error probabilistic complexities of f, respectively, and $D(f)$ the deterministic complexity in the boolean decision tree model. We show the following for any function f.

THEOREM 3.

$$D(f) \leqq 2R_1(f)^2.$$

THEOREM 4.

$$D(f) \leqq 8R_2(f)^3.$$

Impagliazzo and Naor [IN] have considered the uniform analogue of decision trees. Using our results, and paralleling some results in [BI], they showed that, if P = NP, then DTIME(poly-log) = RTIME(poly-log). Here RTIME(t) is the class of problems that can be solved in time t on a randomized TM, even allowing bounded two-sided error.

2. Critical sensitivity and block sensitivity. In this section we will discuss the relationships between several complexity measures of boolean functions. The relationships we show here will then have implications regarding CREW PRAM complexity and boolean decision trees. The complexity measures we consider are the "*critical sensitivity*," the "*block sensitivity*," and the "*certificate*" complexity. We will also mention the relation of these to the boolean decision tree complexity.

NOTATION. Let w be a boolean string of length n, let S be any subset of indices, $S \subset \{1 \cdots n\}$, then $w^{(S)}$ means the string w, with all bits in S flipped. That is, $w^{(S)}$ differs from w exactly on S.

DEFINITION. Let f be a boolean function, w any input string, and i any index. We say f is *sensitive* to x_i on w if $f(w) \neq f(w^{(i)})$. The *critical sensitivity* of f on w, $s_w(f)$, is the number of locations i such that f is sensitive to x_i on w. The *critical sensitivity* of f, $s(f)$, is the maximum over all w of the critical sensitivity of f on w.

This complexity measure of boolean functions has been discussed in the literature under various names. It is sometimes called the "critical complexity" of the function

[W]; sometimes a function with critical sensitivity k is said to have a k-critical input, and is sometimes just called the sensitivity.

Simon [Si] shows that every function that depends on all its variables has critical sensitivity of at least $\Omega(\log n)$. Turan [Tu] showed that all *graph* properties have critical sensitivity of at least $\Omega(v)$. Cook, Dwork, and Reishuk [CDR] use the critical sensitivity of a function to give lower bounds for the CREW PRAM complexity. What we do here is consider a generalization of the critical sensitivity by allowing several bits to be flipped together to change the value of the function.

DEFINITION. Let f be a boolean function, w any boolean string, and S any subset of indices. We say that f is sensitive to S on w if $f(w) \neq f(w^{(S)})$. The *block sensitivity* of f on w, $bs_w(f)$ is the largest number t such that there exists t *disjoint* sets $S_1, S_2, \cdots, S_t$ such that for all $1 \leqq i \leqq t$, f is sensitive to S_i on w. The *block sensitivity* of f, $bs(f)$, is the maximum over all w of the block sensitivity of f on w.

Our main lemma will be the relation between the block sensitivity and the *certificate* complexity.

DEFINITION. Let f be a boolean function, and w any input string. A 1-*certificate* (0-certificate) for f is an assignment to some subset of the variables that forces the value of f to 1 (0). The *certificate complexity of f on w*, $C_w(f)$, is the size of the smallest certificate that agrees with w. The *certificate complexity of f*, $C(f)$, is the maximum over all w of $C_w(f)$.

The certificate complexity of a function describes how many bits of the input must be revealed to you (by someone who knows all the input bits) in order to convince you of the value of the function. This complexity measure has also been widely used in the literature, again under various names. It is sometimes called the sensitivity [VW], [W], sometimes the boolean degree of a function; it can also be viewed as the nondeterministic complexity in the boolean decision tree model. The 1-certificates of f are the terms of f, and the 0-certificates of f are the terms of the complement of f.

We will first mention the obvious relations between them (see [W]) in the following proposition.

PROPOSITION 2.1. *For any f:*

$$s(f) \leqq bs(f) \leqq C(f).$$

Proof. The left inequality follows directly from the definitions. The right inequality follows from the fact that for any input w, any certificate for w must include at least one variable from each set f is sensitive to on w. □

It turns out that for a large subclass of functions these three measures of functions are really equal.

PROPOSITION 2.2. *For all <u>monotone</u> functions f:*

$$s(f) = bs(f) = C(f).$$

Proof. It is enough to show that $C(f) \leqq s(f)$. Consider a minimal certificate of size $C(f)$; without loss of generality assume it is a 1-certificate. The string which has 1 in every bit of the certificate and 0 in all other places, will have critical sensitivity of $C(f)$. The reason is that turning off any of the 1-bits will change the value of the function to 0. □

The following example shows that for general, nonmonotone, functions the inequalities may be strict.

Example 2.3. Let f be the symmetric function on n variables defined to be true if and only if exactly $n/2$ or $(n/2)+1$ of the inputs are 1 (for simplicity, assume that 4 divides n).

It is not difficult to see that the worst case input for all three measures is an input with exactly $n/2$ 1's. Flipping any 1-bit to a 0 will change the value of the function, but flipping any 0-bit to 1 will not change the value, thus the sensitivity is $n/2$. Flipping any two 0-bits to 1 will also change the value of the function, thus by pairing the 0-bits in an arbitrary way we can get another $n/4$ blocks, which gives a block sensitivity of $3n/4$. Finally, any certificate for this input must contain all the 1-bits and all but one of the 0-bits. This gives a certificate complexity of $n-1$.

A function with a quadratic gap between the critical sensitivity and the block sensitivity was found by [Ru]; it is still an open problem whether the gap may be bigger (superpolynomial?). Wegener and Azdori [WZ] exhibit a function with a polynomial (but subquadratic) gap between the critical sensitivity and the certificate complexity. Inspection of their results reveals that they actually give polynomial gaps between the block sensitivity and the certificate complexity. Our main lemma shows that the certificate complexity may only be polynomially bigger than the block sensitivity.

LEMMA 2.4. *For all boolean functions f*:

$$bs(f) \geqq \sqrt{C(f)}.$$

Proof. Let w be an input achieving the certificate complexity, i.e., every certificate for w is of length at least $C(f)$. Let S_1 be some *minimal* set of indices such that $f(w) \neq f(w^{(S_1)})$, let S_2 be another minimal set *disjoint* from S_1, such that $f(w) \neq f(w^{(S_2)})$, and in general we pick S_i to be a minimal set disjoint from all previous sets picked such that $f(w) \neq f(w^{(S_i)})$. We continue picking these sets until at a certain point no such set exists; call the last set S_t.

The union of all sets has to be a certificate for w, since otherwise we could have picked yet another set that changes the value of the function when flipped. Thus we get that:

$$\sum_{i=1}^{t} |S_i| \geqq C(f).$$

Now we can bound the block sensitivity of f in two ways:

(1) f is sensitive to each S_i on w, thus $bs_w(f) \geqq t$.

(2) Since for each i, S_i is minimal, then on $w^{(S_i)}$, f if sensitive to each element in S_i, thus $bs_{w^{(S_i)}}(f) \geqq |S_i|$.

So if $t \geqq \sqrt{C(f)}$, then (1) gives us the desired result; otherwise at least one of the sets has to be of size larger than $\sqrt{C(f)}$ and (2) will give us the result. □

We conclude this section by mentioning the relationship between the certificate complexity, and the boolean decision tree complexity. This result was independently discovered by several people, perhaps first by Blum [Bl].

LEMMA 2.5.

$$C(f) \leqq D(f) \leqq (C(f))^2.$$

We believe that the polynomial relationship between the block sensitivity and the decision tree complexity given by Lemmas 2.4 and 2.5 is interesting in its own right. Sections 3 and 4 give applications of this fact regarding CREW PRAMs and randomized boolean decision trees, and we believe that further application may come. Let us just mention a recent result of Szegedy [Sz].

Szegedy showed that any real polynomial approximating a boolean function f requires a degree of at least $\sqrt{bs(f)}$. Since it is also clear that a real polynomial of degree $D(f)$ can represent the function exactly, we get that the degree required to represent f as a real polynomial is polynomially related to $D(f)$ and to $bs(f)$.

3. PRAM complexity.

3.1. CREW PRAMs. Let $CREW(f)$ denote the parallel time needed to compute f on a CREW PRAM with an unbounded number of processors, each given arbitrary power. [CDR] gave a general lower bound for $CREW(f)$ in terms of the critical sensitivity of f. They showed that for all f:

$$CREW(f) \geqq \log_{\alpha} s(f),$$

where α is some constant less than 5.

We first note that this result may be strengthened to give a bound in terms of the block sensitivity.

LEMMA 3.1. *For all f:*

$$CREW(f) \geqq \log_{\alpha} bs(f).$$

Proof. Let w be an input achieving the block sensitivity, and let $S_1, S_2, \cdots, S_t$ be the sets f is sensitive to on w. We define a new function $f'(X_1, X_2, \cdots, X_t)$ as follows: $f(X_1, \cdots, X_t)$ is equal to $f(w')$ where w' is derived from w by flipping all the bits in the set S_i for each i such that $X_i = 1$. It is easy to see that f' instantly reduces to f on a CREW PRAM, and that the critical sensitivity of f' on the input $000 \cdots 000$ is t. Thus

$$CREW(f) \geqq CREW(f') \geqq \log_{\alpha} s(f') \geqq \log_{\alpha} t = \log_{\alpha} bs(f). \qquad \square$$

The surprising fact is that Lemma 3.1 actually gives a tight lower bound for every function f! That will be shown using decision trees.

LEMMA 3.2. *A* CREW PRAM *can simulate a boolean decision tree of depth d in* $\log_2 d$ *timesteps* (*using* 2^d *processors*).

Proof. We will have a processor for each node of the decision tree. In the first step each processor will read the input variable that belongs to its node and set up a pointer to point to the node that should be followed by this node according to the value of the input. From now on, in each step all the processors will use the standard "pointer doubling" method, and repeatedly copy the pointer of the node they are pointing to into their own pointer. It is easy to see that after $\log_2 d$ steps the root will point to the last leaf reached in the computation. $\square$

The only thing left to note is that the upper and lower bound that we gave are actually to within a constant factor from each other.

THEOREM 1. *For all f:*

$$CREW(f) = \Theta(\log D(f)) = \Theta(\log bs(f)).$$

Proof. Lemmas 2.4 and 2.5 show that $D(f)$ and $bs(f)$ are polynomially related to each other; thus the bounds given in Lemmas 3.1 and 3.2 are within a constant factor of each other. $\square$

It should be noted that the upper bound simulation can be carried out by processors limited to a reasonable instruction set, and on memory cells that contain only 1 bit, while the lower bound derived in [CDR] holds regardless of the instruction set of the processors or the capacity of the memory cells. As a corollary, we get that this model is insensitive to these issues as long as the number of processors is not limited.

3.2. CROW PRAMs. An extra bonus to be derived from the previous proof is the equivalence in computation time between CREW PRAMs and the seemingly weaker CROW PRAMs. (The connection between CROW PRAMs and decision trees was also observed by [Ra].) Let $CROW(f)$ be the time needed to compute f on an ideal CROW PRAM (with an unlimited number of processors) then we get Theorem 2.

THEOREM 2. *For any boolean function* f:

$$CROW(f)=\Theta(CREW(f)).$$

Proof. The simulation of decision trees described in the proof to Lemma 3.2 can also be carried out by a CROW PRAM. □

It is interesting to note that this result does *not* yield a simulation. The relation between CREW PRAM complexity and decision trees is a global one; there does not exist a correspondence between specific stages of the computation and parts of the decision tree. We can actually show that a CROW PRAM cannot simulate a CREW PRAM step in constant time. We exhibit a problem on a *partial* domain that separates these two models. Consider the following "promise" problem: Compute the OR of n input bits when you are "promised" that at most one input bit may be 1. A CREW PRAM with n processors can compute this function in one step; it turns out that a CROW PRAM, however, cannot compute this function quickly. The following lemma appears in [DR] and is attributed there to Snir.

LEMMA 3.3 (DR, Lemma 2.3). *A* CROW PRAM *requires time* $\Omega(\log n)$ *to compute the* OR *function even on this partial domain.*

For completeness, we sketch the proof.

Proof. We say a processor p *depends* on input location i at time t, if its state on the input consisting of all zeros is different than its state on the input consisting of a 1 in location i and zeros everywhere else, at time t. (The processors' state includes its private memory as well as all memory owned by it.)

The following fact is easily proved by induction on t: A processor p at time t depends on at most 2^t locations. The induction step is proved by observing that at time t, a processor p can depend on: (1) any location it depended on at time $t-1$ and (2) any location that the memory cell it read during the last step depended on. The induction hypothesis suffices to bound (2) since these locations are a subset of the locations which the processor that owned the memory cell depended on. The lemma follows since a processor that writes the answer on the all-zeros input must depend on all locations in order to make sure that not one of them is one. □

This lemma does not contradict Theorem 2, since Theorem 2 only holds for functions on *full* domains.

4. Probabilistic versus deterministic decision trees. We first relate the one-sided error randomized complexity of a function to its deterministic complexity.

THEOREM 3. *For any boolean function* f:

$$D(f)\leqq 2(R_1(f))^2.$$

This theorem will follow from the following more general lemma. Let $C^{(1)}(f)$ denote the certificate complexity of f limited to the 1-instances, i.e., the maximum of $C_w(f)$ over all w such that $f(w)=1$.

LEMMA 4.1. *For any boolean function* f:

$$D(f)\leqq 2C^{(1)}(f)R_2(f).$$

Proof. The proof is by induction on the following quantity: $R^*(f)$ is defined to be the smallest integer greater or equal to twice the minimum over all randomized decision trees that compute f with two-sided error of the maximum over all 0-*instances* of f of the expected running time of the randomzed decision tree on the 0-instance. We will build a deterministic decision tree for f of depth at most $C^{(1)}(f)R^*(f)$. This suffices since clearly $R^*(f)\leqq 2R_2(f)$.

In the base case $R^*(f)=0$, which implies that a randomized algorithm exists which makes no queries on 0-instances, which implies that the function is constant. In the induction steps the deterministic algorithm for f will start by picking any 1-certificate of f of size at most $C^1(f)$ and asking all the variables in it. After this first step we are left with an induced function f' on the remaining variables, and the algorithm will recursively solve this induced problem. Our claim is that (1) $C^{(1)}f' \leqq C^{(1)}f$ and (2) $R^*(f') \leqq R^*(f)-1$. The lemma follows since by the construction $D(f) \leqq C^{(1)}(f)+D(f')$.

The fact that the $C^{(1)}$ complexity cannot increase for an induced subfunction is obvious; we now prove the second claim. Consider the randomized algorithm for f running on any 0-instance, w. The probability that this algorithm queries some variable in the 1-certificate must be at least $\frac{1}{2}$. The reason is that without querying some bit in the 1-certificate, the algorithm cannot distinguish between w, for which f is 0, and w with all the bits in the 1-certificate flipped to conform with the 1-certificate for which f is 1.

It follows that f' can be solved by a randomized decision tree that makes an expected $\frac{1}{2}$ of a query less than the one for f made on any 0-instance (this is done by taking the randomized decision tree for f and erasing any query that was already asked). Thus the R^* complexity decreases by at least one. □

Proof of Theorem 3. Theorem 3 is an immediate corollary of Lemma 4.1 since both $R_2(f)$ and $C^{(1)}(f)$ are lower bounds for $R_1(f)$. The first bound follows since any one-sided error decision tree can be made into a two-sided error decision tree by simply accepting with probability $\frac{1}{4}$ at every reject leaf (this increases the acceptance probability for any 1-instance from $\frac{1}{2}$ to $\frac{3}{4}$). The second bound follows since a one-sided error decision tree must always be correct on 0-instances and thus must see a 1-certificate before accepting. □

The previous techniques do not carry over to the two-sided error case. In order to give results here, we will need to use our results concerning the block sensitivity. We first show that the block sensitivity can serve as a lower bound for the two-sided error randomized complexity of a function.

LEMMA 4.2. *For all boolean functions f:*

$$R_2(f) \geqq \frac{bs(f)}{2}.$$

Proof. Let w be the input that achieves the block sensitivity, and let $S_1, S_2, \cdots, S_t$ be disjoint sets such that f is sensitive to S_i on w. For each $1 \leqq i \leqq t$, any randomized algorithm running on w must query some variable in S_i with probability of at least $\frac{1}{2}$, since otherwise it cannot distinguish between w and $w^{(S_i)}$. Thus the total expected time has to be at least $(t/2)$. □

At this point we immediately get the following theorem.

THEOREM 4. *For any function f:*

$$D(f) \leqq 8R_2(f)^3.$$

Proof. From Lemma 4.1 an upper bound for $D(f)$ is $2C(f)R_2(f)$; using Lemma 2.4 this is bounded from above by $2bs(f)2R_2(f)$. An application of Lemma 4.2 completes the proof of the theorem. □

5. Acknowledgments. I would like to thank Russell Impagliazzo for his contributions to the paper; Amos Fiat, Valerie King, and Moni Naor for many helpful discussions; and Avi Wigderson for pointing out the application to CROW PRAMs.

REFERENCES

[BI] M. BLUM AND R. IMPAGLIAZZO, *Generic oracles and oracle classes*, in Proc. 28th ACM Symposium on Foundations of Computer Science, 1987.

[CDR] 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.

[DR] P. W. DYMOND AND W. L. RUZZO, *Parallel* RAM*s with owned global memory and deterministic context-free language recognition*, ICALP, 1986.

[IN] R. IMPAGLIAZZO AND M. NAOR, *Decision trees and downward closures*, Structure in Complexity Conference, 1988.

[K] V. KING, *The randomized complexity of graph properties*, manuscript, 1987.

[KK] D. J. KLEITMAN AND D. J. KWIATKOWSKI, *Further results on the Aanderaa-Rosenberg conjecture*, J. Combin. Theory Ser. B, 28 (1980), pp. 85-95.

[KSS] J. KAHN, M. SAKS, AND D. STURTEVANT, *A topological approach to evasiveness*, Combinatorica 4 (1984), pp. 297-306.

[Ra] P. RAGDE, personal communication.

[Ro] A. L. ROSENBERG, *On the time required to recognize properties of graphs: A problem*, SIGACT News 5 #4, 1973.

[Ru] D. RUBINSTEIN, personal communication,

[RV] R. RIVEST AND S. VUILLEMIN, *On recognizing graph properties from adjacency matrices*, Theoret. Comput. Sci., 3 (1976/1977), pp. 371-384.

[Si] H. U. SIMON, *A tight* $\Omega(\log \log n)$ *bound on the time for parallel* RAM'*s to compute nondegenerate boolean functions*, FCT'83, Lecture Notes in Computer Science 158, 1983.

[Sn] M. SNIR, *Lower bounds on probabilistic linear trees*, Theoret. Comput. Sci., 38 (1985), pp. 69-82.

[Sz] M. SZEGEDY, personal communication.

[SW] M. SAKS AND A. WIGDERSON, *Probabilistic boolean decision trees and the complexity of evaluating game trees*, in Proc. 27th ACM Symposium on Foundations of Computer Science, 1986.

[Ta] G. TARDOS, *Query complexity, or Why is it difficult to separate* $NP^A \cap coNP^A$ *from* P^A *by a random oracle A*, manuscript, 1987.

[Tu] G. TURAN, *The critical complexity of graph properties*, Inform. Process. Lett., 18 (1984), pp. 151-153.

[VW] U. VISHKIN AND A. WIGDERSON, *Tradeoffs between depth and width in parallel computation*, SIAM J. Comput., 14 (1985), pp. 303-314.

[W] I. WEGENER, *The complexity of Boolean functions*, John Wiley & Sons, New York, 1987, pp. 373-410.

[WZ] I. WEGENER AND L. AZDORI, *A note on the relation between critical and sensitive complexity*, manuscript.

[Y1] A. YAO, *Probabilistic computations: Towards a unified measure of complexity*, in Proc. 18th ACM Symposium on Foundations of Computer Science, 1977.

[Y2] ———, *Lower bounds to randomized algorithms for graph properties*, in Proc. 28th ACM Symposium on Foundations of Computer Science, 1987.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1008-1020, December 1991
© 1991 Society for Industrial and Applied Mathematics
002

ON THE EXACT COMPLEXITY OF STRING MATCHING: LOWER BOUNDS*

ZVI GALIL† AND RAFFAELE GIANCARLO‡

Abstract. This paper provides several lower bounds on the number of character comparisons that any string matching algorithm must perform in the worst case in order to find occurrences of a pattern string in a text string. The class of algorithms that are considered need not know the alphabet.

Key words. string matching, string searching, text editing, computational complexity, worst case behavior

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U15

1. Introduction. Given a text $t[1, n]$ and a pattern $p[1, m]$, both being strings over an alphabet Σ, *string matching* consists of finding all occurrences of the pattern in the text. Due to its relevance to many application areas, the problem has been extensively studied and many linear, i.e., $O(n+m)$, time algorithms are available. For a survey, the reader is referred to [1]. Many of these algorithms acquire knowledge about the text only through comparisons of the form "$t[i]=p[j]$?" and therefore they work for general alphabets. That is, they need not know the alphabet (it may even be infinite) to work correctly and they have a running time independent of the alphabet size. For this kind of algorithm, we provide several lower bounds for the number of character comparisons that they must perform in the worst case. The best previously known lower bound is the trivial one, i.e., n. Note that if Σ is a binary alphabet, one can use the failure function used by Knuth, Morris, and Pratt [5] to derive deterministic finite automata [2], which perform string matching in exactly n comparisons. Therefore, n is a tight bound for binary alphabets. From now on, we consider only the case $|\Sigma| \geqq 3$.

We recall that Rivest [6] showed that any string matching algorithm must *examine*, in the worst case, $n-m+1$ text characters to find an occurrence of the pattern in the text. A text character is *examined* when the algorithm knows to which character of the alphabet it corresponds. Rivest's model is not comparable to ours. On one hand, it is stronger, as one character examination involves possibly many character comparisons. On the other hand, his algorithms must have prior knowledge of the alphabet.

Our model of computation is a RAM with uniform cost criterion [2] slightly modified. We assume that the text $t[1, n]$ is stored in a special random access buffer. This buffer can be accessed only through a server that is not part of the algorithm executing on the RAM. The server accepts only queries of the form "$p[i]=t[j]$?" and "$t[i]=t[j]$?" and it provides an answer to each request in one unit of time. We count only the comparisons of the form just specified. Note that, on our model, any string matching algorithm can have knowledge about the pattern, but it can acquire knowledge about the relationship between characters in the text and characters in the pattern only

* Received by the editors October 11, 1990; accepted for publication (in revised form) February 6, 1991.

† Department of Computer Science, Columbia University, New York, New York 10027, and Tel-Aviv University, Tel-Aviv, Israel. This author's work was supported in part by National Science Foundation grant CCR 88-14977.

‡ Dipartimento di Matematica, Università di Palermo, Palermo, Italy. Present address, AT&T Bell Laboratories, Murray Hill, New Jersey 07974. This work was performed while the author was visiting the Department of Computer Science, Columbia University, New York, New York 10027. This work was supported in part by National Science Foundation grant CCR 88-14977 and by the Italian Ministry of University and Scientific Research, Project "Algoritmi e Sistemi di Calcolo."

through comparisons with the pattern. Alternatively, we could use a binary decision tree model.

We say that a string matching algorithm is *on-line* if the portion of the text about which it can ask questions is limited to a sliding window of size m. Moreover, the window can be aligned with $t[i, i+m-1]$ if and only if the algorithm has already decided whether an occurrence of the pattern can start at position k of the text, for all k, $1 \leqq k < i$. Let $c(n, m)$ denote the maximal number of character comparisons made by a string matching algorithm, given a text of length n and a pattern of length m over a general alphabet. We approximate it by $(1+C)n$, where C is a universal constant. We add the subscript "*on-line*" when we restrict attention to on-line algorithms and the superscript 1 when we consider algorithms that find only one occurrence of the pattern in the text. The results of this paper can be summarized as follows:

(i) For $0 < m \leqq 2$, $c_{on\text{-}line}(n, m) \geqq n$ for infinitely many n's. For $m \geqq 3$ and odd,

$$c_{on\text{-}line}(n, m) \geqq n + \left\lfloor \frac{2(n-m)}{m+3} \right\rfloor$$

for infinitely many n's. For $m \geqq 3$ and even,

$$c_{on\text{-}line}(n, m) \geqq n + \left\lfloor \frac{2(n-m)}{m+4} \right\rfloor$$

for infinitely many n's.

(ii) For $0 < m \leqq 3$, $c^1_{on\text{-}line}(n, m) \geqq n$ for infinitely many n's. For $m \geqq 4$,

$$c^1_{on\text{-}line}(n, m) \geqq n + \left\lfloor \frac{n-m}{m} \right\rfloor$$

for infinitely many n's. This bound applies only to algorithms that perform comparisons of the form "$t[i] = p[j]$?".

(iii) For $0 < m \leqq 2$, $c(n, m) \geqq n$ for infinitely many n's. For $m \geqq 3$, $c(n, m) \geqq n + \lfloor n/2m \rfloor$ for infinitely many n's.

In a companion paper [4], we have shown that $C \leqq C_{on\text{-}line} \leqq \frac{1}{3}$ and that $C^1_{on\text{-}line} \leqq \frac{1}{3}$. It follows that $\frac{1}{6} \leqq C \leqq C_{on\text{-}line} = \frac{1}{3}$, $\frac{1}{4} \leqq C^1_{on\text{-}line} \leqq \frac{1}{3}$. Therefore, we obtain a tight bound of $\frac{4}{3}n$ comparisons for on-line algorithms that find all occurrences of the pattern in the text.

The lower bounds are general enough to apply even to randomized algorithms that use character comparisons to perform string matching. We use an "unrestricted" adversary that need not know the strategy of the algorithm beforehand. It fixes the pattern and then the text is chosen adaptively according to the questions asked by the algorithm and in such a way as to force the algorithm to make mistakes, i.e., to compare certain text characters twice.

The remainder of this paper is organized as follows. In § 2, we provide lower bounds for $c_{on\text{-}line}(n, m)$ and $c^1_{on\text{-}line}(n, m)$. In § 3, we consider off-line algorithms, i.e., algorithms that have access to the whole text rather than being restricted to a sliding window of size m, and we provide a bound for $c(n, m)$. The last section contains some open problems.

2. On-line algorithms.

THEOREM 1. *Assume that* $|\Sigma| \geqq 3$. *For* $0 < m \leqq 2$, $c_{on\text{-}line}(n, m) \geqq n$ *for infinitely many* n*'s. For* $m \geqq 3$ *and* m *odd (even, respectively),*

$$c_{on\text{-}line}(n, m) \geqq n + \left\lfloor \frac{2(n-m)}{m+3} \right\rfloor \qquad \left(c_{on\text{-}line}(n, m) \geqq n + \left\lfloor \frac{2(n-m)}{m+4} \right\rfloor, \textit{ respectively}\right)$$

for infinitely many n*'s.*

Proof. If $m=1$ ($m=2$, respectively), the adversary chooses the text $t[1, n]=a^n$ and the pattern $\hat{p}[1]=a$ ($\hat{p}[1,2]=aa$, respectively). Any algorithm must perform n comparisons. Thus, the bound follows.

Assume $m \geqq 3$. We choose a pattern $\hat{p}[1, m]=a^l b a^l$ for m odd ($\hat{p}[1, m]=a^l b a^{l+1}$ for m even, respectively). We show that there exists a sequence of texts of increasing length for which the lower bound holds. The adversary fixes the text on-line, as the algorithm compares more text characters. The first step of our construction, for text of length m, is as follows: The adversary sets $t[1, m]=\hat{p}[1, m]$. Any on-line algorithm must perform at least m comparisons between text and pattern characters in order to discover the occurrence at position 1. Therefore, the lower bound holds.

Now assume that there exists a text $t[1, n_0]$ having $\hat{p}[1, m]$ as suffix, for which the lower bound holds and for which the algorithm already knows all occurrences of the pattern in the text. Assume also that no comparison has been made beyond n_0. We extend $t[1, n_0]$ to a text $t[1, n_1]$ having $\hat{p}[1, m]$ as suffix, for which the lower bound holds and no comparison has been made beyond n_0. We extend $t[1, n_0]$ to a text $t[1, n_1]$ having $\hat{p}[1, m]$ as suffix, for which the lower bound holds and no comparison has been made beyond n_1.

Since the algorithm already knows all occurrences of the pattern in $t[1, n_0]$ and $t[1, n_0]$ has $\hat{p}[1, m]$ as suffix, $t[n_0-l+1, n_0]=a^l$ and $t[n_0-l]=b$ ($t[n_0-l]=a$, respectively). Since $t[n_0-l]=b$ does not match any character in $\hat{p}[1, l]$ ($t[n_0-l, n_0]=a^{l+1}$ does not match $\hat{p}[l+1]=b$, respectively), the pattern can be shifted past text position n_0-l. Moreover, since $t[n_0-l+1, n_0]=a^l$, an occurrence of the pattern can start at n_0-l+1. The on-line algorithm can ask questions involving text characters in $t[n_0-l+1, n_0+l+1]$ ($t[n_0-l+1, n_0+l+2]$, respectively). Since the algorithm knows that $t[n_0-l+1, n_0]=a^l$ and we are interested in finding a lower bound, we can assume it will not ask questions involving these text positions.

The adversary fixes $t[n_0+3, n_0+l+1]=a^{l-1}$ ($t[n_0+3, n_0+l+2]=a^l$, respectively) and each question asked by the algorithm about these text positions is answered consistently with that choice. Note that, if $l=1$, $t[n_0+3, n_0+l+1]$ is the empty string. The adversary fixes text characters $t[n_0+1, n_0+2]$ as well as n_1 and $t[n_1]$ according to the first question that the algorithm asks about these text positions. This is done as follows.

(1) The algorithm asks first "$t[n_0+1]=b$?" or "$t[n_0+2]=a$?". The adversary fixes $n_1=n_0+l+2$ ($n_1=n_0+l+3$, respectively), $t[n_0+1, n_0+2]=ab$, and $t[n_1]=a$. Therefore, the answer to the question is no. The algorithm can rule out text position n_0-l+1, since no occurrence can be there. But it cannot rule out text position n_0-l+2, since $t[n_0-l+2, n_1]=a^l b a^l$ ($t[n_0-l+2, n_1]=a^l b a^{l+1}$, respectively). The algorithm must ask $l+2$ ($l+3$, respectively) questions involving $t[n_0+1, n_1]=t[n_0+1, n_0+l+2]$ ($t[n_0+1, n_1]=t[n_0+1, n_0+l+3]$, respectively) to find such an occurrence of the pattern in the text. Therefore, the total number of questions involving text positions in $t[n_0+1, n_1]$ is $l+3$ ($l+4$, respectively). Since the algorithm has made at least

$$n_0+\left\lfloor \frac{2(n_0-m)}{m+3} \right\rfloor \qquad \left(n_0+\left\lfloor \frac{2(n_0-m)}{m+4} \right\rfloor, \text{ respectively}\right)$$

character comparisons for $t[1, n_0]$, adding $l+3$ ($l+4$, respectively) to this bound and recalling that $m=2l+1$ ($m=2l+2$, respectively) and that $n_1=n_0+l+2$ ($n_1=n_0+l+3$, respectively), we obtain a lower bound of

$$n_1+\left\lfloor \frac{2(n_1-m)}{m+3} \right\rfloor \qquad \left(n_1+\left\lfloor \frac{2(n_1-m)}{m+4} \right\rfloor, \text{ respectively}\right)$$

for text $t[1, n_1]$. Moreover, such text has the pattern as suffix and no comparison has been made beyond n_1.

(2) The algorithm asks first "$t[n_0+2]=b$?" or "$t[n_0+1]=a$?". We fix $t[n_0+1, n_0+2]=ba$ and $n_1=n_0+l+1$ ($n_1=n_0+l+2$, respectively). Therefore, the answer to the question is no. There is an occurrence of the pattern in the text starting at n_0-l+1, and any algorithm must perform at least $l+1$ ($l+2$, respectively) character comparisons involving $t[n_0+1, n_1]=t[n_0, n_0+l+1]$ ($t[n_0+1, n_1]=t[n_0, n_0+l+2]$, respectively) to find it. Therefore, the total number of questions involving text positions in $t[n_0+1, n_1]$ is at least $l+2$ ($l+3$, respectively). Since the algorithm has made at least

$$n_0+\left\lfloor\frac{2(n_0-m)}{m+3}\right\rfloor \qquad \left(n_0+\left\lfloor\frac{2(n_0-m)}{m+4}\right\rfloor, \text{ respectively}\right)$$

character comparisons for $t[1, n_0]$, adding $l+2$ ($l+3$, respectively) to this bound and recalling that $m=2l+1$ ($m=2l+2$, respectively) and that $n_1=n_0+l+1$ ($n_1=n_0+l+2$, respectively), we obtain a lower bound of (more than)

$$n_1+\left\lfloor\frac{2(n_1-m)}{m+3}\right\rfloor \qquad \left(n_1+\left\lfloor\frac{2(n_1-m)}{m+4}\right\rfloor, \text{ respectively}\right)$$

for text $t[1, n_1]$. Moreover, such text has the pattern as suffix and no comparison has been made beyond n_1.

(3) The algorithm asks first "$t[n_0+1]=t[j]$?" or "$t[n_0+2]=t[j]$?", $j\leqq n_0+l+1$. We discuss only the case "$t[n_0+1]=t[j]$?" since "$t[n_0+2]=t[j]$?" can be handled similarly. We consider three subcases.

—$j\leqq n_0$: If $t[j]=b$, then a proof that the bound holds can be obtained as in (1). If $t[j]=a$, then a proof that the bound holds can be obtained as in (2).

—$j=n_0+2$: The adversary sets $t[n_0+1, n_0+2]=ab$. The bound follows by a reasoning analogous to the one in (1).

—$n_0+2<j\leqq n_0+l+1$: Since the adversary sets $t[n_0+3, n_0+l+1]=a^{l-1}$, the bound follows by a reasoning analogous to the one in (2).

Now, the theorem follows inductively. □

Recall that a string $x[1, m]$ is periodic of period i if and only if i is the minimal integer such that $x[1, m-i]=x[i+1, m]$. If so such $i<m$ exists, $x[1, m]$ is said to be nonperiodic.

We observe that for $m=1, 2, 3$, the lower bounds of Theorem 1 are tight for any pattern. Indeed, the algorithm by Colussi [3] performs at most

$$n+\left\lfloor(n-m)\left(\frac{z'}{z+z'}\right)\right\rfloor\leqq n+\left\lfloor\frac{n-m}{2}\right\rfloor$$

character comparisons [4] for any pattern of length $m=z+z'$ and period $z>z'$. For any nonperiodic pattern of length 2 or 3, $z'=0$ and the algorithm by Colussi performs n comparisons to find all occurrences of those patterns in any given text of length n. For $\hat{p}[1, m]=a^m$, we can find all occurrences of $\hat{p}[1]=a$ and then determine where $\hat{p}[1, m]$ occurs in the text with no extra character comparisons. Since we can find all occurrences of $\hat{p}[1]=a$ in a text of length n in n comparisons ($z'=0$), a bound of n holds also for $\hat{p}[1, m]=a^m$, $1\leqq m\leqq 3$. The last pattern we have to consider is $\hat{p}[1, m]=aba$. Since $z'=1$ and $m=3$, the upper bound of Colussi's algorithm and the lower bound of Theorem 1 are equal. Therefore, $c_{\textit{on-line}}(n, m)=n$ for $m=1, 2$ and $c_{\textit{on-line}}(n, m)=n+\lfloor(n-m)/3\rfloor$ for $m=3$.

Recall that $c^1_{\textit{on-line}}(n, m)$ denotes the minimum number of comparisons that any on-line algorithm must perform in order to find the leftmost occurrence of the pattern in the text. We prove a lower bound for algorithms that perform only character

comparisons of the form "$p[i]=t[j]$?" and consider only this kind of algorithm for the remainder of this section. The *Hamming distance* $d(x, y)$ between two strings x and y of equal length is defined as the number of positions with mismatching characters in the two strings. Given a string x, let Σ_x be the set of characters of Σ that appear in x. Assume that $|\Sigma_x|<|\Sigma|$. Consider an algorithm that tests whether $d(x, y)=1$, where x is given and can be preprocessed while y is specified on-line. Assume that the algorithm knows at the beginning of the computation that x and y match in exactly k specified positions. Moreover, for each $y[j]$, it also knows a proper subset of Σ_x of size $\hat{r}_j<|\Sigma_x|$ such that $y[j]$ and $x[j]$ are not characters in that subset.

LEMMA 1. *There is an adversary that can choose y such that $d(x, y)=1$ and force the algorithm to perform at least $|x|-k+r_q-\hat{r}_q-1$ character comparisons, where $r_q>\hat{r}_q$ is the number of distinct characters of Σ_x that cannot be assigned to the unique $y[q]\neq x[q]$ without violating the answers given to the questions of the algorithm and its initial knowledge about y. Moreover, the algorithm does not know which character of Σ is $y[q]$.*

Proof. Let $G=(V, E)$ be a bipartite graph with vertex partition $V_1=\{1, 2, \cdots, |x|\}$ and $V_2=\Sigma$. E has two kinds of edges, *red* and *blue*. An edge (i, a) is red if and only if $y[i]=x[i]=a$. An edge (i, a) is blue if and only if $y[i]\neq a$. A red edge (blue, respectively) (i, a) is initially in E if and only if the algorithm knows that $y[i]=a$ ($y[i]\neq a$, respectively). Initially, the graph has k red edges. The adversary answers the questions of the algorithm and updates E. In doing so, it also specifies which characters must be assigned to y. Its strategy consists of two parts.

Part 1: While the number of red edges is less than $|x|-1$, the question "$y[i]=x[h]$?" is answered "yes" and $y[i]$ is set to $x[i]$ if and only $x[h]=x[i]$. The edge $(i, x[h])$ is added to E, if it is not already there. Its color is red if the answer is "yes" and blue otherwise.

The first part adds at least $|x|-1-k$ red edges. Let q be the only node of V_1 that has no red edge in common with a node of V_2. We now consider only questions involving $y[q]$. (The other symbols in y have been determined.)

Part 2: For each question "$y[q]=x[h]$?", the answer is "no" and the blue edge $(q, x[h])$ is added to E, if it is not already there.

Let r_q be the number of blue edges incident on q after step 2. Note that since the algorithm can ask only questions involving a character in each string, it must eventually ask "$y[q]=x[h]$?", where $x[h]=x[q]$. After this question is asked, it can stop, since it knows that $d(x, y)=1$. This question also causes a new blue edge with endpoint in q to be added to E. Thus, the second part adds $r_q-\hat{r}_q\geqq 1$ blue edges. Therefore, the algorithm has made at least $|x|-k+r_q-\hat{r}_q-1$ comparisons to discover that $d(x, y)=1$. Moreover, for each blue edge (q, a), $a\in\Sigma_x$, $y[q]\neq a$, and $y[q]$ cannot be assigned r_q characters of Σ_x. (It may be assigned some character in $\Sigma-\Sigma_x$, since $|\Sigma_x|<|\Sigma|$.) Since $y[q]$ has been involved only in comparisons with a negative outcome and the algorithm did not know which character of Σ is $y[q]$ when it started, it still does not have this information. □

Let $v[1, m]=a^l b a^{l+1}$ ($v[1, m]=a^l b a^{l+2}$, respectively), $l\geqq 1$, and let w be a given string such that $d(v, w)=1$, where $v[j]\neq w[j]$. Let w' be the longest suffix of w that matches a prefix of v.

LEMMA 2. *When $j<m$ and $w[j]=c\neq a, b$, $w'=a^{l-i+1}$, where $i=\max(1, j-l-1)$ ($i=\max(1, j-l-2)$, respectively). When $j=m$ and $w[m]=b\neq a=v[m]$, $w'=a^l b$ and w' is empty when $j=m$ and $w[m]=c\neq a, b$.*

Proof. Assume that $j<m$ and $w[j]=c\neq a, b$. When $j\leqq l+1$, $w'=a^l$, and when $j>l+1$, $w'=a^{2l-j+2}$ ($w'=a^{2l-j+3}$, respectively). Therefore, $w'=a^{l-i+1}$, where $i=\max(1, j-l-1)$ ($i=\max(1, j-l-2)$, respectively). The other cases are obvious. □

THEOREM 2. *Assume that* $|\Sigma| \geqq 3$ *and that the algorithm can perform only character comparisons of the form* "$t[h]=p[j]$?". *For* $0<m\leqq 3$, $c^1_{on\text{-}line}(n, m)\geqq n$ *for infinitely many n's. For* $m \geqq 4$,

$$c^1_{on\text{-}line}(n, m)\geqq n+\left\lfloor\frac{n-m}{m}\right\rfloor$$

for infinitely many n's.

Proof. For $m=1, 2, 3$, we fix $\hat{p}[1, m]=a, ab, abb$, respectively. Note that all the chosen patterns are nonperiodic, i.e., there is no integer $i<m$ such that $\hat{p}[1, m-i]=\hat{p}[i+1, m]$. Assume that there exists a text $t[1, n_0]$ for which the lower bound holds and that satisfies the constraints: There is no occurrence of the pattern in $t[1, n_0-m+1]$ and the least integer for which an occurrence can start is n_0+1. Moreover, no comparison has been made beyond n_0.

Note that for $n_0=0$, the empty text satisfies those constraints. We extend $t[1, n_0]$ to a text $t[1, n_1]$ for which the lower bound holds and that satisfies the same constraints as $t[1, n_0]$. The adversary chooses $n_1=n_0+m$. Since the least integer for which an occurrence can start is n_0+1, the algorithm must test whether $t[n_0+1, n_1]=\hat{p}[1, m]$. By Lemma 1, $t[n_0+1, n_1]$ can be chosen such that $d(t[n_0+1, n_1], \hat{p}[1, m])=1$ and such that the algorithm must ask at least m questions to compute d (since it knows nothing about $t[n_0+1, n_1]$). Moreover, we can fix $t[n_1-m+j]=c\in\Sigma-\{a, b\}$, where $t[n_1-m+j]\neq\hat{p}[j]$. Since the pattern is not periodic, no c occurs in it, and $t[n_0+1, n_1]\neq\hat{p}[1, m]$, there is no occurrence of the pattern in $t[1, n_1-m+1]$ and the least integer for which an occurrence can start is n_1+1. Moreover, no comparison has been made beyond n_1.

We now assume that $m\geqq 4$. We set $\hat{p}[1, m]=a^lba^{l+1}$ ($\hat{p}[1, m]=a^lba^{l+2}$, respectively), $l\geqq 1$. The proof idea here is similar to the one used in Theorem 1 for the case $m\geqq 3$: We have a text $t[1, n_0]$ for which the bound holds and we extend it to a text $t[1, n_1]$ for which the bound also holds. n_1 and $t[n_0+1, n_1]$ must be chosen so that the algorithm is "fooled" at least once. However, there are some differences here that make the proof more involved. In extending $t[1, n_0]$ to $t[1, n_1]$, the adversary in some cases must force the algorithm to make at least two mistakes to rule out occurrences of the pattern in two contiguous text positions. To achieve this goal, the adversary takes advantage of the fact that when the algorithm has discovered that $t[k]\neq\hat{p}[j]$ and kills an occurrence of the pattern in the text, the algorithm does not know which character of Σ is $t[k]$.

Assume that there exists a text $t[1, n_0]$ that contains no occurrence of the pattern and that satisfies the following constraints:

(a) $d(t[n_0-m+1, n_0], \hat{p}[1, m])=1$; the algorithm does not know which character of Σ is $t[n_0-m+j]\neq\hat{p}[j]$ and no comparison has been made beyond n_0.

(b) If $j<m$ or $j=m$ and if $t[n_0]\neq a$ but may be equal to b, then

$$c^1_{on\text{-}line}(n_0, m)\geqq n_0+\left\lfloor\frac{n_0-m}{m}\right\rfloor.$$

(c) If $j=m$ and $t[n_0]\neq a, b$ then $c^1_{on\text{-}line}(n_0, m)\geqq n_0+\lfloor n_0/m\rfloor$.

Note that for $n_0=m$, Lemma 1 guarantees that there is a text satisfying those constraints. Indeed, let $x=\hat{p}[1, m]$ and $y=t[1, m]$ in Lemma 1. The algorithm knows nothing about y. By Lemma 1, y can be picked such that $d(x, y)=1$. Let $y[j]\neq x[j]$. The algorithm does not know which character of Σ is $y[j]$ and no comparison has

been made beyond $n_0 = m$. Thus, (a) is satisfied. If $r_j = 1$, the algorithm has performed at least n_0 comparisons and knows only that $y[j] \neq a$ (when $x[j] = a$) or that $y[j] \neq b$ (when $x[j] = b$). Thus (b) is satisfied when $j < m$ or $j = m$ and when $t[m] \neq a$ but may be equal to b. If $r_j = 2$, the algorithm has performed at least $n_0 + 1$ comparisons and it knows that $t[j] \neq a, b$. Thus (c) holds when $j = m$ and $t[m] \neq a, b$.

We extend $t[1, n_0]$ to a text $t[1, n_1]$ that contains no occurrence of the pattern and that satisfies the same constraints as $t[1, n_0]$. We consider three cases: $j < m$ and $t[n_0 - m + j] \neq \hat{p}[j]$; $j = m$ and $t[n_0] \neq a$, but it may be equal to b; and $j = m$ and $t[n_0] \neq a, b$.

(1) $t[n_0 - m + j] \neq \hat{p}[j]$ and $j < m$. We first show that $t[n_0 - m + j]$ can be fixed so that no occurrence of the pattern in the text can start in $t[n_0 - m + 2, n_0 - l + i - 1]$, where $i = \max(1, j - l - 1)$ ($i = \max(1, j - l - 2)$, respectively). Indeed, since (a) holds for $t[1, n_0]$, $t[n_0 - m + j]$ is not known to the algorithm and can be any character $c \in \Sigma - \{a, b\}$. The adversary sets $t[n_0 - m + j] = c$ and tells it to the algorithm for free. By Lemma 2 (with $w = t[n_0 - m + 1, n_0]$), $t[n_0 - l + i, n_0] = a^{l-i+1}$ is the longest suffix of $t[n_0 - m + 1, n_0]$ to match a prefix of $\hat{p}[1, m]$. Therefore, the next candidate position for an occurrence is $n_0 - l + i$. Since the algorithm knows nothing about $t[n_0 + 1, n_0 + l + i + 1]$ ($t[n_0 + 1, n_0 + l + i + 2]$, respectively) and it is on-line, it must ask questions about these text positions in order to rule out $n_0 - l + i$ as an occurrence. If the adversary could allow an occurrence of the pattern in the text, it would set $t[n_0 + 1, n_0 + i - 1] = a^{i-1}$, $t[n_0 + i + 2, n_0 + l + i + 1] = a^l$ ($t[n_0 + i + 2, n_0 + l + i + 2] = a^{l+1}$, respectively), and it would answer the questions about these text characters consistently with this choice. Then it would choose $t[n_0 + i, n_0 + i + 1]$ and n_1 according to the first question asked about these characters and in such a way as to force the algorithm to make a mistake. Unfortunately, the adversary cannot allow an occurrence of the pattern in the text. Thus it cannot fix $t[n_0 + 1, n_0 + i - 1]$ and $t[n_0 + i + 2, n_0 + l + i + 1]$ ($t[n_0 + i + 2, n_0 + l + i + 2]$, respectively) a priori, but it must fix these text characters on-line, as the algorithm keeps asking questions about them. Assume that the algorithm has asked e questions about $t[n_0 + 1, n_0 + i - 1]$ and $t[n_0 + i + 2, n_0 + l + i + 1]$ ($t[n_0 + i + 2, n_0 + l + i + 2]$, respectively) and it has discovered that $e' \leqq e$ of these characters are "a." (The strategy of the adversary is to answer "yes" only to questions "$t[j] = a$," for any j such that $n_0 + 1 \leqq j \leqq n_0 + i - 1$ or $n_0 + i + 2 \leqq j \leqq n_0 + l + i + 1$ ($n_0 + i + 2 \leqq j \leqq n_0 + l + i + 2$, respectively).) Let the $e + 1$st question be about $t[n_0 + i, n_0 + i + 1]$. The algorithm asks:

(1.1) "$t[n_0 + i] = a$?" or "$t[n_0 + i + 1] = b$?". We discuss only the case "$t[n_0 + i] = a$?", since the case "$t[n_0 + i + 1] = b$?" can be handled similarly. The answer to the question is "no." Let $start = n_0 - l + i$. The adversary sets $n_1 = n_0 + l + i + 1 = start + m - 1$ ($n_1 = n_0 + l + i + 2 = start + m - 1$, respectively). Let $y[1, m] = t[start, n_1]$ and $x[1, m] = \hat{p}[1, m]$. As in Lemma 1, at the time the question "$y[l + 1] = t[n_0 + i] = a$?" is answered, the algorithm already knows that $k = e' + (n_0 - start + 1)$ characters of y match the corresponding characters of x. Moreover, $\sum_{f=l-i+2}^{m} \hat{r}_f = e - e' + 1$ since the algorithm has asked $e + 1$ questions about $t[n_0 + 1, n_1] = y[l - i + 2, m]$ and it has discovered e' matches with $x[l - i + 2, m]$. In particular, $\hat{r}_{l+1} = 1$ since $t[n_0 + i] \neq a$ and this is the first comparison for $t[n_0 + i]$. By Lemma 1, the $m - k$ characters of y not known to the algorithm can be chosen (on-line) so that $d(y, x) = 1$ and so that the algorithm is forced to ask at least $m - k + r_{j'} - \hat{r}_{j'} - 1$ questions to discover that $d(y, x) = 1$, where $r_{j'}$ is the number of symbols in $\{a, b\}$ that cannot be assigned to $y[j'] \neq x[j']$. Moreover, the algorithm does not know which character of Σ is $y[j']$. The algorithm has asked e preliminary questions about $t[n_0 + 1, n_0 + i - 1]$ and $t[n_0 + i + 2, n_0 + l + i + 1]$ ($t[n_0 + i + 2, n_0 + l + i + 2]$, respectively), "$y[l + 1] = t[n_0 + i] = a$?". Thus the number of questions asked to rule out an occurrence at $start = n_0 - l + i$ is $e + (m - k + r_{j'} - \hat{r}_{j'} - 1) + 1 =$

$n_1 - n_0 + r_{j'} - \hat{r}_{j'} + e - e'$. Therefore, $c^1_{on\text{-}line}(n_1, m) \geqq c^1_{on\text{-}line}(n_0, m) + n_1 - n_0 + r_{j'} - \hat{r}_{j'} + e - e'$. Since we are considering the case $j < m$, we can use the bound in (b) and obtain

$$\text{(I)} \qquad c^1_{on\text{-}line}(n_1, m) \geqq n_1 + \left\lfloor \frac{n_0 - m}{m} \right\rfloor + r_{j'} - \hat{r}_{j'} + e - e'.$$

By Lemma 1, $r_{j'} - \hat{r}_{j'} \geqq 1$. Since $e - e' \geqq 0$, $n_1 = n_0 + l + i + 1$ ($n_1 = n_0 + l + i + 2$, respectively), $m = 2l + 2$ ($m = 2l + 3$, respectively), and (I), we obtain that

$$c^1_{on\text{-}line}(n_1, m) \geqq n_1 + \left\lfloor \frac{n_1 - m}{m} \right\rfloor,$$

for $j' < m$. Thus, if $j' < m$ or $j' = m$ and $t[n_1] \neq a$ but it may be equal to b, $t[1, n_1]$ satisfies the bound in (b). Assume now that $j' = m$ and $t[n_1] = y[m] \neq a, b$. Note that $r_m = 2$. Since $e - e' + 1 = \sum_{f=l-i+2}^{m} \hat{r}_f \geqq \hat{r}_{l+1} + \hat{r}_m$ and $\hat{r}_{l+1} = 1$, $e - e' \geqq \hat{r}_m$. Using the identities $r_m = 2$, $n_1 = n_0 + l + i + 1$ ($n_1 = n_0 + l + i + 2$, respectively), $m = 2l + 2$ ($m = 2l + 3$, respectively), the inequality $e - e' \geqq \hat{r}_m$ and bound (I), we obtain that $c^1_{on\text{-}line}(n_1, m) \geqq n_1 + \lfloor n_1/m \rfloor$. Thus $t[1, n_1]$ satisfies the bound in (c). It also follows by Lemma 1 that the algorithm does not know which character of Σ is $t[n_1 - m + j']$ and no comparison has been made beyond n_1, so (a) holds for $t[1, n_1]$. Obviously, there is no occurrence of the pattern in $t[1, n_1]$.

(1.2) "$t[n_0 + i] = b$?" or "$t[n_0 + i + 1] = a$?". The answer to the question is "no." Therefore there can be no occurrence of the pattern at $n_0 - l + i$, but there may be one at $n_0 - l + i + 1$ since $t[n_0 - l + i, n_0] = a^{l-i+1}$. Let $start = n_0 - l + i + 1$. The adversary sets $n_1 = n_0 + l + i + 2 = start + m - 1$ ($n_1 = n_0 + l + i + 3 = start + m - 1$, respectively). Using arguments similar to the ones in (1.1), we can show that it is possible to choose $t[start, n_1]$ so that $t[1, n_1]$ contains no occurrence of the pattern and it satisfies constraints (a), (b), and (c).

(2) $j = m$, $t[n_0] \neq \hat{p}[m] = a$, but $t[n_0]$ may be equal to b. We first show that no occurrence of the pattern can start in $[n_0 - m + 2, n_0 - l - 1]$. Let $v = \hat{p}$ and $w = t[n_0 - m + 1, n_0]$ in Lemma 2. Indeed, if $t[n_0] = c$, by Lemma 2 the next candidate for an occurrence is $n_0 + 1$. If $t[n_0] = b$, by Lemma 2, the next candidate for an occurrence is $n_0 - l$. Since $t[n_0]$ is not known to the algorithm and the algorithm is on-line, it must test the minimum of those two candidates, i.e., $n_0 - l$. The adversary sets $n_1 = n_0 + l + 1 = start + m - 1$ ($n_1 = n_0 + l + 2 = start + m - 1$, respectively). Let $y[1, m] = t[start, n_1]$ and $x[1, m] = \hat{p}[1, m]$. As in Lemma 1, the algorithm knows that $k = n_0 - start$ characters of y match the corresponding characters of x. Moreover, $\hat{r}_{l+1} = 1$ since $y[l+1] = t[n_0] \in \Sigma - \{a\}$ and $\sum_{f=l+2}^{m} \hat{r}_f = 0$, since no comparison has been made beyond n_0. By Lemma 1, the $m - k$ characters of y not known to the algorithm can be chosen (on-line) so that $d(y, x) = d(t[start, n_1], \hat{p}[1, m]) = 1$ and so that the algorithm is forced to ask at least $m - k + r_{j'} - \hat{r}_{j'} - 1$ questions to discover that $d(y, x) = 1$, where $r_{j'}$ is the number of symbols in $\{a, b\}$ that cannot be assigned to $y[j'] = t[start + j' - 1] \neq \hat{p}[j']$. Moreover, the algorithm does not know which character of Σ is $t[start + j' - 1]$. Thus the algorithm has asked at least $m - k + r_{j'} - \hat{r}_{j'} - 1 = n_1 - n_0 + r_{j'} - \hat{r}_{j'}$ questions (all about $t[n_0, n_1]$) to rule out an occurrence at $start = n_0 - l$. Since we are assuming that $j = m$, $t[n_0] \neq \hat{p}[m] = a$ and $t[n_0] \in \Sigma - \{a\}$, the bound in (b) holds for $c^1_{on\text{-}line}(n_0, m)$, i.e.

$$c^1_{on\text{-}line}(n_0, m) \geqq n_0 + \left\lfloor \frac{n_0 - m}{m} \right\rfloor,$$

and we obtain bound (I) for $c^1_{on\text{-}line}(n_1, m)$, with

$$e - e' = \sum_{f=l+2}^{m} \hat{r}_f = 0.$$

We can use the exact same arguments following bound (I) in case (1.1) (with $i=0$) to show that $t[1, n_1]$ satisfies constraints (a), (b), and (c) also in the case being considered here.

(3) $j=m$, $t[n_0] \neq \hat{p}[m]=a$ and $t[n_0] \neq a, b$. Since $t[n_0]=c$, by Lemma 2 (with $w=t[n_0-m+1, n_0]$), the next candidate position is $start=n_0+1$. The adversary sets $n_1=n_0+2l+2=start+m-1$ ($n_1=n_0+2l+3=start+m-1$, respectively). Let $y[1, m]=t[start, n_1]$ and $x[1, m]=\hat{p}[1, m]$. The algorithm knows nothing about y since no comparison has been made beyond n_0. By Lemma 1, y can be chosen (on-line) so that $d(y, x)=d(t[start, n_1], \hat{p}[1, m])=1$ and so that the algorithm is forced to ask at least $m+r_{j'}-\hat{r}_{j'}-1$ questions to discover that $d(y, x)=1$, where $r_{j'}$ is the number of symbols in $\{a, b\}$ that cannot be assigned to $y[j']=t[start+j'-1] \neq \hat{p}[j']$. Thus, the algorithm has asked at least $m+r_{j'}-\hat{r}_{j'}-1=n_1-n_0+r_{j'}-\hat{r}_{j'}-1$ questions (all about $t[n_0, n_1]$) to rule out an occurrence at $start=n_0+1$. Since we are assuming that $j=m$, $t[n_0] \neq \hat{p}[m]=a$ and $t[n_0] \in \Sigma-\{a, b\}$, the bound in (c) holds for $c^1_{on\text{-}line}(n_0, m)$, i.e.,

$$c^1_{on\text{-}line}(n_0, m) \geqq n_0 + \left\lfloor \frac{n_0}{m} \right\rfloor$$

and we obtain

$$\text{(II)} \qquad c^1_{on\text{-}line}(n_1, m) \geqq n_1 + \left\lfloor \frac{n_0}{m} \right\rfloor + r_{j'} - \hat{r}_{j'} - 1.$$

By Lemma 1, $r_{j'}-\hat{r}_{j'} \geqq 1$. Since $n_1=n_0+m$ and (II), we obtain that

$$c^1_{on\text{-}line}(n_1, m) \geqq n_1 + \left\lfloor \frac{n_1-m}{m} \right\rfloor,$$

for $j' \leqq m$. Thus, if $j'<m$ or $j'=m$ and $t[n_1] \neq a$ but it may be b, $t[1, n_1]$ satisfies the bound in (b). Assume now that $j'=m$ and $t[n_1]=y[m] \neq a, b$. Note that $r_m=2$. Recall that $\hat{r}_m=0$. Using the identities $r_m-\hat{r}_m=2$, $n_1=n_0+m$, and (II), we obtain that $c^1_{on\text{-}line}(n_1, m) \geqq n_1+\lfloor n_1/m \rfloor$. Thus $t[1, n_1]$ satisfies the bound in (c). It also follows by Lemma 1 that the algorithm does not know which character of Σ is $t[n_1-m+j']$ and no comparison has been made beyond n_1, so (a) holds for $t[1, n_1]$. Obviously, there is no occurrence of the pattern in $t[1, n_1]$.

Now the theorem follows inductively. □

We observe that for $m=1, 2, 3$, the bound of Theorem 2 is tight for any pattern. For $m=1, 2$, our claim follows from the fact that $c_{on\text{-}line}(n, m)=n$. For $m=3$, we have already shown that if $p[1, 3]=aaa$ or $p[1, 3]$ is not periodic, then the Colussi algorithm can find all of its occurrences in any text of length n in at most n comparisons. Thus we have to show that it is possible to find the leftmost occurrence of $p[1, 3]=aba$ in n comparisons. In the pseudocode given below, i is a potential start of an occurrence of aba in $t[1, n]$.

```
    begin
i ← 1; found ← false;
while i ≦ n − 2 and found = false do
  begin
    if t[i+1] ≠ b then i ← i+1;
                  else
                    begin
                      if t[i] ≠ a then i ← i+2;
                                 else
```

```
                                   begin
                                    if t[i+2] ≠ a then i ← i+3;
                                               else found ← true;
                                   end
end
    end
   end
```

Let $j > 1$ be the smallest integer such that $t[j] = b$ in $t[1, n]$. Note that the algorithm performs $j-1$ character comparisons to find it since $t[1]$ is not compared. If the jth comparison is not successful, the next occurrence can start at $t[j+1, n]$ and the algorithm tries to find the leftmost occurrence of aba in $t[j+1, n]$ and it has performed at most j comparisons. If it is successful, the $j+1$st comparison will either cause the algorithm to stop or the algorithm tries to find the leftmost occurrence of aba in $t[j+2, n]$ and, in any case, it has performed at most $j+1$ comparisons. By induction, we get that the algorithm performs at most n comparisons to find out whether aba occurs in $t[1, n]$.

3. Off-line algorithms. We now consider algorithms that can ask questions of the form "$t[i] = t[j]$?" or "$t[i] = p[k]$?" and have access to the whole text rather than being restricted to a sliding window of size m.

THEOREM 3. *Assume that* $|\Sigma| \geqq 3$. *For* $0 < m \leqq 2$, $c(n, m) \geqq n$ *for infinitely many* n*'s. For* $m \geqq 3$, $c(n, m) \geqq n + \lfloor n/2m \rfloor$ *for infinitely many* n*'s.*

Proof. The case $m = 1, 2$ is as in Theorem 1 and is therefore omitted. So assume $m \geqq 3$. The adversary chooses $\hat{p}[1, m] = a^l b a^l (\hat{p}[1, m] = a^l b a^{l+1}$, respectively) for m odd (even, respectively), $l \geqq 1$. The adversary chooses $n = m(2q+1)$, $q \geqq 0$.

The text is divided in $2q+1$ consecutive blocks, each of size m. These blocks are numbered $0, 1, \cdots, 2q$. The ith block is $t[im+1, (i+1)m]$. For each even-numbered block, the adversary sets the corresponding text characters to $a^l b a^l$ ($a^l b a^{l+1}$, respectively). This will force any algorithm to perform at least $m(q+1)$ comparisons between text characters in the even blocks and pattern or text characters. (Recall that the algorithm can ask questions of the form "$t[i] = t[j]$?" or "$t[i] = p[k]$?".)

We concentrate only on the questions that the algorithm asks in order to find occurrences of the pattern in the text starting or ending in positions in odd-numbered blocks.

The adversary puts in each odd block either the string $a^l b a^l$ ($a^l b a^{l+1}$, respectively) or $a^{l-1} b a^l b$ ($a^{l-1} b a^l ba$, respectively) depending on the questions that the algorithm asks during its execution (we specify later how this is actually done).

Note that if $a^l b a^l$ is placed in an odd block, an occurrence of the pattern starts in the first position of that block. Given that even blocks $t[(i-1)m+1, im] = t[(i+1)m+1, (i+2)m] = a^l b a^l$ ($a^l b a^{l+1}$, respectively), the placement of $a^{l-1} b a^l b$ ($a^{l-1} b a^l ba$, respectively) in $t[im+1, (i+1)m]$ completes two occurrences of the pattern in the text, one starting at im and the other at $im + l + 1$. For each odd block $t[im + 1, (i+1)m]$, the adversary sets $t[im+1, im+l-1] = t[im+l+2, im+2l] = a^{l-1}$ (and $t[(i+1)m] = a$, respectively), since, irrespective of which string is placed in $t[im + 1, (i+1)m]$, those parts of $t[im+1, (i+1)m]$ are equal to a^{l-1} (a^{l-1} and a, respectively). Each question about those text characters is answered consistently with the choice of the adversary. The placement strategy of a string in an odd block is such that the algorithm must first waste at least one question on that block and then it can find the occurrences of the pattern that start or end in the block. Therefore, the algorithm receives at least one "no" and m "yes" answers for each odd block. From now on,

we will describe the placement strategy and count the "no" answers that the algorithm receives.

We say that $t[k]$ is *specified* if the adversary has assigned a character to it. Similarly, an odd block $t[im+1,(i+1)m]$ is *specified* when the adversary has placed either $a^l ba^l$ ($a^l ba^{l+1}$, respectively) or $a^{l-1}ba^l b$ ($a^{l-1}ba^l ba$, respectively) in it. Otherwise, it is *unspecified.* Let $\circ$ denote the concatenation operation. Let $x^i[1,3] = t[im+l, im+l+1] \circ t[im+2l+1]$, i odd. For each position h in $t[im+l, im+l+1] \circ t[im+2l+1]$ let $\hat{h}$ be its corresponding position in $x^i[1,3]$. Note that specifying an odd block consists of setting either $x^i = aba$ or $x^i = bab$, since the remaining characters of the block have already been specified. Note also that if $x^i = x^j$, i and j odd, then $x^i[s] = x^j[g]$ if and only if $s \equiv g \pmod 2$. Similarly, if $x^i \neq x^j$, then $x^i[s] = x^j[g]$ if and only if $s \not\equiv g \pmod 2$. Unless otherwise stated, from now on, we refer to odd-numbered blocks simply as blocks. While assigning one of the two strings to blocks and answering questions, the adversary keeps the following invariant.

INVARIANT 1. *At the time $t[im+1,(i+1)m]$, i odd, is specified, the algorithm has received a "no" for at least one question involving characters in the block and knows nothing about which characters of the pattern appear in x^i.*

Initially, this invariant is trivially satisfied. The strategy of the adversary can be outlined as follows. It dynamically maintains an undirected acyclic graph that encodes the negative answers to the questions that the algorithm asked about unspecified blocks. Such answers impose constraints on which string has to be placed in a block. Each additional question involving two unspecified text characters in unspecified blocks is answered consistently with the graph and then the graph is possibly modified to include this new question. Each question of the form "$t[h] = \hat{p}[r]$?" or "$t[h] = t[k]$?", $t[h]$ unspecified and $t[k]$ specified, is answered so that the algorithm makes a mistake. Then, the adversary specifies some blocks, including the one in which $t[h]$ is. As is shown later, this is equivalent to deleting a connected component from the graph. Each node in the graph corresponds to an unspecified block and each edge corresponds to a "no" to a question involving two unspecified characters in two unspecified blocks. Each node in a connected component is assigned one of two colors by the adversary. Each connected component has a different pair of colors. When the adversary has enough information to specify a block, all blocks having its color in its connected component will be assigned the same string. The remaining blocks in the connected component will be assigned the other string (recall that the adversary has two strings to choose from). Initially, the graph is composed of q nodes; each is colored differently and is not connected to any other. The adversary maintains the graph and answers the questions asked by the algorithm according to the following policy:

(1) The algorithm asks "$t[h] = t[k]$?", where both text characters are unspecified and they belong to unspecified blocks. Thus, $h \in \{im+l, im+l+1, im+2l+1\}$ and $k \in \{jm+l, jm+l+1, jm+2l+1\}$, for some i and j odd.

(1.a) $t[h]$ and $t[k]$ belong to blocks whose nodes are in the same connected component and have the same color. The adversary answers "yes" if and only if $\hat{h} \equiv \hat{k} \pmod 2$. Since x^i and x^j will both either get *aba* or *bab*, the answer is consistent.

(1.b) $t[h]$ and $t[k]$ belong to blocks whose nodes have different colors but are in the same connected component. The adversary answers "yes" if and only if $\hat{h} \not\equiv \hat{k} \pmod 2$. Since x^i will be assigned *aba* (or *bab*) while x^j will be assigned *bab* (or *aba*), the answer is consistent.

(1.c) $t[h]$ and $t[k]$ belong to blocks that are in different connected components of the graph. The adversary answers "no." Let C_1 and C_2 be the two connected components and let R_1 (R_2) be the set of nodes of C_1 (C_2) having the same color as

the node of $t[h]$ ($t[k]$). C_1 and C_2 must be united and the nodes of C_2 must be consistently "painted" using the colors of C_1. It is done as follows:

* $\hat{h} \not\equiv \hat{k} \pmod 2$: Blocks in R_1 and R_2 must be assigned the same string. Since x^i and x^j will both either get *aba* or *bab*, the answer is consistent. The graph must be changed to account for this fact. All the nodes of R_2 are "painted" with the color of R_1. The remaining nodes in C_2 are "painted" with the other color in C_1. An edge is added between the blocks in R_1 and R_2 to which $t[h]$ and $t[k]$ belong, respectively.

* $\hat{h} \equiv \hat{k} \pmod 2$: Blocks in R_1 and R_2 must be assigned different strings. Since x^i will be assigned *aba* (or *bab*) while x^j will be assigned *bab* (or *aba*), the answer is consistent. The graph must be modified. All the nodes not in R_2 are "painted" with the color of R_1. The nodes in R_2 are "painted" with the other color in C_1. An edge is added between the blocks in R_1 and R_2 to which $t[h]$ and $t[k]$ belong, respectively.

(2) The algorithm asks "$t[h]=\hat{p}[r]$?" or "$t[h]=t[k]$?", where $t[k]$ is specified (it may belong to an unspecified block). The answer to the question is "no." The adversary specifies which of the two strings must be assigned to the blocks represented by nodes in the same connected component of the node representing block $t[im+1,(i+1)m]$. Moreover, such a component is removed from the graph. This is done as follows:

(2.a) $\hat{p}[r]=a$. (The same procedure is applied to the case $t[k]=a$.)

* $\hat{h}$ is odd: Block $t[im+1,(i+1)m]$ is set equal to $a^{l-1}ba^lb$ ($a^{l-1}ba^lba$, respectively). Therefore, $x^i = bab$ and the assignment is consistent with the answer given. Each block having a node of the same color as the node of $t[im+1,(i+1)m]$ gets the same string. The remaining blocks in the component get a^lba^l (a^lba^{l+1}, respectively). The component is removed from the graph.

* $\hat{h}=2$: Block $t[im+1,(i+1)m]$ is set equal to a^lba^l (a^lba^{l+1}, respectively). Therefore, $x^i = aba$ and the assignment is consistent with the answer given. Each block having a node of the same color as the node of $t[im+1,(i+1)m]$ gets the same string. The remaining blocks in the component get $a^{l-1}ba^lb$ ($a^{l-1}ba^lba$, respectively). The component is removed from the graph.

(2.b) $\hat{p}[r]=b$ (the same procedure is applied to the case $t[k]=b$): It is the complement of (2.a) with respect to the parity of $\hat{h}$.

(3) The algorithm asks "$t[h]=t[k]$?", where both text characters are specified (they may belong to unspecified blocks). The answer is "yes" if and only if $t[h]=t[k]$.

Assume that Invariant 1 is satisfied immediately after a connected component is removed from the graph. Then we show that it is satisfied immediately after the next component is removed from the graph: Observe that once a connected component of r nodes is removed from the graph, the algorithm has received at least r negative answers to questions involving characters in the blocks belonging to the component ($r-1$ correspond to edges in (1.c) and one corresponds to the question asked in (2)). At the time $t[vm+1,(v+1)m]$, v odd, is specified, the algorithm knows nothing about which characters of the pattern appear in x^v, since those text characters either have not been compared with pattern characters or the comparison is as in (2.a) and it has received a negative answer. Therefore, the total number of negative answers is at least q. Adding the $m(2q+1)$ positive answers that the algorithm must receive to find all occurrences of the pattern in the text ($m(q+1)$ for even blocks and mq for odd blocks), we obtain that $c(n,m) \geqq m(2q+1)+q = n+\lfloor n/2m \rfloor$ since $n=m(2q+1)$ and $q=\lfloor (2q+1)/2 \rfloor$. □

4. Concluding remarks and open problems. We have shown lower bounds for $c_{on\text{-}line}(n,m)$, $c^1_{on\text{-}line}(n,m)$, and $c(n,m)$, respectively. Combining the results of this

paper with the ones of [4], we have that $\frac{1}{6} \leqq C \leqq C_{on\text{-}line} = \frac{1}{3}$ and $\frac{1}{4} \leqq C^1_{on\text{-}line} \leqq \frac{1}{3}$. Moreover, $C_{on\text{-}line} = \frac{1}{3}$ is the best possible. A number of open problems remain. It would be interesting to obtain the exact value of C. It is still a challenge to close the gap between the upper bounds reported in [4] and lower bound presented here for $c(n, m)$, $c_{on\text{-}line}(n, m)$ and $c^1_{on\text{-}line}(n, m)$, respectively. These gaps grow as m grows. Note that our lower bound for $c^1_{on\text{-}line}(n, m)$ holds only for algorithms that perform comparisons of the form "$t[i] = p[j]$?". It would be interesting to extend it to algorithms that are also allowed comparisons of the form "$t[i] = t[j]$?". Note that we do not have a lower bound for $c^1(n, m)$, the number of comparisons for off-line algorithms that search for only one occurrence of the pattern in the text.

REFERENCES

[1] A. V. AHO, *Algorithms for finding pattern in strings*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier, Amsterdam, 1990, pp. 257–295.

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

[3] L. COLUSSI, *Correctness and efficiency of string matching algorithms*, Inform. and Comput., to appear.

[4] Z. GALIL AND R. GIANCARLO, *On the exact complexity of string matching: Upper bounds*, SIAM J. Comput., 21 (1992), to appear.

[5] D. E. KNUTH, J. H. MORRIS, AND V. B. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 189–195.

[6] R. V. RIVEST, *On the worst case behavior of string-searching algorithms*, SIAM J. Comput., 6 (1977), pp. 669–674.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1021–1033, December 1991
© 1991 Society for Industrial and Applied Mathematics
003

SELF-P-PRINTABILITY AND POLYNOMIAL TIME TURING EQUIVALENCE TO A TALLY SET*

ROY S. RUBINSTEIN†

Abstract. The class of self-P-printable sets, those sets that are enumerable in polynomial time relative to themselves, has been shown to be the same as the class of sets that have small generalized Kolmogorov complexity relative to themselves [J. Hartmanis and L. Hemachandra, *Proc. 3rd Annual Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag, Berlin, New York, 1986, pp. 321–333] and [J. Balcázar and R. Book, *Acta Informatica*, 23 (1986), pp. 679–688]. This class properly includes the sets of small generalized Kolmogorov complexity, including the tally sets. The class of sets polynomial time Turing equivalent to a tally set includes the class of self-P-printable sets [J. Balcázar and R. Book, *op. cit.*], and the inclusion is clearly proper because every self-P-printable set must be sparse while there are many nonsparse sets (such as all nonsparse sets in P) that are polynomial time Turing equivalent to a tally set.

The question of whether or not the class of self-P-printable sets is the same as the class of *sparse* sets that are polynomial time Turing equivalent to a tally set is addressed here. Necessary and sufficient conditions for the equivalence of these two classes are presented. Also presented are relativizations for all possible combinations of these necessary and sufficient conditions, suggesting that the nonrelativized solution to the question of this equivalence will be difficult to determine.

Key words. P-printable sets, tally sets, sparse sets, Kolmogorov complexity, relativization, computational complexity

AMS(MOS) subject classifications. 03D15, 68Q15, 68Q30, 68Q05

1. Introduction. Computational complexity theory is the study of the quantitative aspects of computing. In particular, the time and space needed for the recognition of various sets has been of primary importance. This study has produced, among many other things, the definitions of the complexity classes P and NP, the polynomial hierarchy, and the theory of NP-completeness.

Associated with this is the study of various structural properties of sets. These structural properties have known important relationships to computational complexities, but they do not correlate exactly. Thus, a direct correspondence between structural properties and computational complexity does not always hold. We will call classes of sets defined by these properties *structural complexity classes.* The most well known of the structural complexity classes are the classes of sparse and tally sets, both of which cut across all computational complexity classes. A summary of results concerning these may be found in [21].

Other structural complexity classes that have been studied include the classes of sets based on generalized Kolmogorov complexity, the P-printable sets, the self-P-printable sets, and the class of sets polynomial time Turing equivalent to a tally set [2], [4], [9], [10], [11], [12], [20].

It has been shown that the self-P-printable sets are precisely those sets that have small self-relativized generalized Kolmogorov complexity [10], [4], and that the class of sets polynomial time Turing equivalent to a tally set is the same as the class of sets having self-producible circuits [4]. It is stated (correctly, though without proof) in [4] that the class of self-P-printable sets is properly included in the class of sets that

* Received by the editors June 22, 1990; accepted for publication (in revised form) January 21, 1991. These results were obtained while the author was at the College of Computer Science, Northeastern University, Boston, Massachusetts 02115, and was supported by National Security Agency grant MDA904-87-H-2020.

† Computer Science Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, Massachusetts 01609.

have self-producible circuits (and therefore are polynomial time Turing-equivalent to a tally set). These two classes are shown to be different by showing a nonsparse set that has self-producible circuits and noting that a self-P-printable set must be sparse. In fact, all sets in P are polynomial time Turing equivalent to the empty set and therefore have self-producible circuits, regardless of density.

The question of whether considering only *sparse* sets that are polynomial time Turing equivalent to a tally set would still yield a proper inclusion then needs to be addressed. That is precisely what this paper does. Section 3 includes the following results.

THEOREM 1.1. *If there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable, then there exists a tally set T such that there is a sparse set in* $\mathrm{FewP}^T - \mathrm{P}^T$.

This theorem has the following corollary.

COROLLARY 1.2. *If there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable, then* $\mathrm{P} \neq \mathrm{NP}$.

A partial converse to the above theorem is also found, giving us a sufficient condition for the existence of a sparse set that is polynomial time Turing equivalent to a tally set but not self-P-printable.

THEOREM 1.3. *If there exists a sparse set in* $\mathrm{FewP} - \mathrm{P}$*, then there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable.*

Section 4 provides relativizations involving FewP, suggesting that it would be difficult to solve the nonrelativized question of whether or not all sparse sets that are polynomial time Turing equivalent to a tally set are self-P-printable. This includes an oracle relative to which $\mathrm{P} \neq \mathrm{FewP}$ and there are no sparse sets in $\mathrm{FewP} - \mathrm{P}$, an oracle relative to which there exist sparse sets in $\mathrm{FewP} - \mathrm{P}$, and others.

2. Preliminaries. It is assumed here that the reader has basic familiarity with the standard notions and classes in complexity theory.

We use the standard lexicographic ordering $\leq$ on strings, and $|w|$ denotes the length of the string w. All strings here are elements of $\{0,1\}^*$, and all sets are subsets of $\{0,1\}^*$. A tally language is a subset of $\{1\}^*$. The cardinality of a set S is denoted $\|S\|$. $A \subset B$ denotes $A \subseteq B$ and $A \neq B$.

Standard polynomial time pairing functions are used, and the pairing of strings x and y, for example, is denoted $\langle x, y\rangle$. The pairing functions will have the property that $|\langle x, y\rangle| \leq 2(|x| + |y|)$, that $\langle x, y\rangle$ can be determined from x and y in polynomial time, and that x and y can each be determined from $\langle x, y\rangle$ in polynomial time. This same notation, when used for grouping more than two strings, actually denotes successive applications of the pairing function. All logarithms are base 2.

PSV is the class of functions computable deterministically in polynomial time. For any set A, PSV^A is the class of functions computable deterministically in polynomial time using A as an oracle.

DEFINITION 2.1. A set A is *sparse* if there exists a polynomial p such that the number of strings in A of length less than or equal to n is less than or equal to $p(n)$.

Generalized Kolmogorov complexity, a two-parameter version of Kolmogorov complexity that includes information about not only how far a string can be compressed, but how fast it can be restored, was introduced by Hartmanis ([9]), whose definition is presented here.

DEFINITION 2.2. For a (deterministic) Turing machine M and functions g and G mapping natural numbers to natural numbers, let

$$K_M[g(n), G(n)] = \{x \mid (\exists y)[|y| \leq g(|x|) \text{ and } M(y) = x \text{ in } G(|x|) \text{ or fewer steps}]\}.$$

It was shown in [9] that there exists a universal Turing machine M_u such that for any other Turing machine M there exists a constant c such that

$$K_M[g(n), G(n)] \subseteq K_{M_u}[g(n) + c, cG(n) \log G(n) + c].$$

Dropping the subscript, $K[g(n), G(n)]$ will actually denote $K_{M_u}[g(n), G(n)]$. This relativizes in a straightforward manner, where

$$K^A[g(n), G(n)] = \{x \mid (\exists y)[|y| \leq g(|x|) \text{ and } M_u^A(y) = x \text{ in } G(|x|) \text{ or fewer steps}]\}.$$

DEFINITION 2.3. A set is said to have *small generalized Kolmogorov complexity* if it is a subset of $K[c \log n, n^c]$ for some c.

DEFINITION 2.4. A set S is *polynomial time printable* (P*-printable*) if there exists a polynomial p such that all the elements of S of length less than or equal to n can be printed by a deterministic machine in time $p(n)$.

P-printability relativizes (as in a set being P^A-printable) by allowing the printing machine to use an oracle. A set is *self*-P*-printable* if it is P-printable relative to itself.

For any finite set A, we let $c(A)$ denote an encoding of A. It is assumed that for any string x and finite set A, computing $c(A \cup \{x\})$ from $c(A)$ and x and deciding if $x \in A$ from $c(A)$ and x can both be done in time polynomial in $|c(A)| + |x|$. For any set A and natural number n, let $A^{\leq n}$ denote the set of all strings in A of length less than or equal to n. Then, $c(A^{\leq n})$ denotes the encoding of an initial segment of A.

DEFINITION 2.5. For any set A, $enum_A$ is the function that, for each n, on input 0^n, produces $c(A^{\leq n})$.

Note that $enum_A \in \mathrm{PSV}$ is equivalent to A being P-printable, and that $enum_A \in \mathrm{PSV}^A$ is equivalent to A being self-P-printable.

Allender ([1], [2]) defined the complexity class FewP, a subclass of NP, as follows.

DEFINITION 2.6. FewP is the class of languages that are accepted by nondeterministic polynomial time Turing machines M for which there is a polynomial p such that for all inputs w, there are fewer than $p(|w|)$ accepting computations of M on w.

UP [6], [22], [8] is the class of languages in FewP which are accepted by nondeterministic polynomial time Turing machines with unique accepting computations. Both UP and FewP are subclasses of NP defined by restricting the number of accepting computations. Densities of accepting computations were previously considered in [18].

We say that a function f *majorizes* a function g if for all but finitely many n, $f(n) > g(n)$.

Sets having self-producible circuits, a restricted version of P/poly, were introduced in [14]:

DEFINITION 2.7. A set A has *self-producible circuits* if there exist a polynomial length-bounded function $h: \{0\}^* \to \{0,1\}^*$ and a set B in P such that the following conditions hold:

(1) for every $x \in \{0,1\}^*$, $x \in A \iff \langle x, h(0^{|x|}) \rangle \in B$;
(2) the function h can be computed relative to A in deterministic polynomial time.

Note that without the second condition, the class defined would be the same as P/*poly*.

For the remainder of this paper, unless otherwise stated, S will denote a sparse set and T will denote a tally set.

3. Self-P-printability and polynomial time Turing equivalence to a tally set. Self-P-printability, a variation of P-printability, has close ties to relativized generalized Kolmogorov complexity and other structural complexities. It was shown in [10], [4] that a set is self-P-printable if and only if it has small generalized Kolmogorov complexity relative to itself.

Sets having self-producible circuits, a restricted version of P/poly, were introduced in [14] and are studied in [4], where it is shown that the sets with self-producible circuits are precisely those that are polynomial time Turing equivalent to some tally set. Reference [4] states without proof that the class of self-P-printable sets is properly included in the class of sets with self-producible circuits.

The following theorem shows the inclusion.

THEOREM 3.1. *Every self-*P*-printable set has self-producible circuits.*

Proof. Let S be a P^S-printable set with h the enumerating function (i.e., h with oracle S is $enum_S$ and operates in polynomial time). Let $B = \{\langle x, y\rangle \mid y$ is a list of strings of length less than or equal to $|x|$ and x is in the list$\}$. Clearly $B \in \mathrm{P}$. Now $x \in S \iff \langle x, h(0^{|x|})\rangle \in B$, so S has self-producible circuits. □

COROLLARY 3.2. *Every self-*P*-printable set is polynomial time Turing equivalent to some tally set.*

This inclusion is clearly proper, as every self-P-printable set must be sparse, and there are many nonsparse sets that are polynomial time Turing equivalent to a tally set. All sets in P, regardless of density, are polynomial time Turing equivalent to the empty set, for example.

The question naturally arises as to whether or not the class of sets with self-producible circuits that are also sparse coincides with the class of self-P-printable sets. Theorems 3.4 and 3.6 will suggest that this would be difficult to prove either way. Several proofs will use the relativization of a result by Allender, so it will be stated here as a proposition.

PROPOSITION 3.3 ([1], [2]). *For any set A, there is a sparse set in P^A which is not P^A-printable if and only if there is a sparse set in $\mathrm{FewP}^A - \mathrm{P}^A$.*

We will now compare the class of self-P-printable sets with the class of sparse sets that are polynomial time Turing equivalent to a tally set.

THEOREM 3.4. *If there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable, then there exists a tally set T such that there is a sparse set in $\mathrm{FewP}^T - \mathrm{P}^T$.*

Proof. Let S be a sparse set that is polynomial time Turing equivalent to a tally set T and that is not self-P-printable. If there is a sparse set in $\mathrm{FewP}^S - \mathrm{P}^S$, then there is a sparse set in $\mathrm{FewP}^T - \mathrm{P}^T$, and we are done.

Assume then that there are no sparse sets in $\mathrm{FewP}^S - \mathrm{P}^S$. By Proposition 3.3, our assumption that there are no sparse sets in $\mathrm{FewP}^S - \mathrm{P}^S$, and the fact that S is a sparse set in P^S, S is P^S-printable. This is a contradiction, so there are sparse sets in $\mathrm{FewP}^S - \mathrm{P}^S$, and therefore there exist sparse sets in $\mathrm{FewP}^T - \mathrm{P}^T$. □

COROLLARY 3.5. *If there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable, then* $\mathrm{P} \neq \mathrm{NP}$.

Proof. Let S be a sparse, non-self-P-printable set that is polynomial time Turing equivalent to a tally set T. By Theorem 3.4 this implies that there exist sparse sets in $\mathrm{FewP}^T - \mathrm{P}^T$, so $\mathrm{P}^T \neq \mathrm{FewP}^T$ and $\mathrm{P}^T \neq \mathrm{NP}^T$. By a result in [17], this implies that $\mathrm{P} \neq \mathrm{NP}$. □

Corollary 3.5 was previously proven in [4] by other methods, and Theorem 3.4 improves upon this. A partial converse is also obtained.

THEOREM 3.6. *If there exists a sparse set in* FewP − P, *then there is a sparse set that is polynomial time Turing equivalent to a tally set and is not self-*P*-printable.*

Proof. The contrapositive will be proven here. Every set in P is polynomial time Turing equivalent to $\emptyset$, so if every sparse set that is polynomial time Turing equivalent to a tally set is self-P-printable, then every sparse set in P is self-P-printable and therefore P-printable. This implies, by Proposition 3.3, that there are no sparse sets in FewP − P. □

The only thing preventing Theorems 3.4 and 3.6 from being converses of one another is that Theorem 3.4 contains the condition "there exists a tally set T such that there is a sparse set in $\mathrm{FewP}^T - \mathrm{P}^T$," while Theorem 3.4 contains the condition "there exists a sparse set in FewP − P."

If we had a theorem that said "There are no sparse sets in FewP − P if and only if for every tally set T, there are no sparse sets in $\mathrm{FewP}^T - \mathrm{P}^T$," then the two conditions would be equivalent, and thus Theorems 3.4 and 3.6 would be converses. Unfortunately, the techniques of [17] used to prove "P = NP if and only if for every tally set T, $\mathrm{P}^T = \mathrm{NP}^T$" do not appear to work for the existence of sparse sets or for FewP.

Theorems 3.4 and 3.6 are easily seen to be true converses, however, while restricting our attention to sparse sets in P as follows.

PROPOSITION 3.7. *There exists a sparse set in* P *that is polynomial time Turing equivalent to a tally set and is not* (*self-*)P*-printable if and only if there exists a sparse set in* FewP − P.

Proof. This follows directly from Proposition 3.3 with the observation that every set in P is polynomial time Turing equivalent to the empty set. □

4. Relativizations of FewP. Theorems 3.4 and 3.6 depend upon the relation between P and FewP—whether they are the same or, if they are different, whether there exist sparse sets in their difference. This section presents oracles relative to which each of these possibilities hold. This suggests that the nonrelativized versions of these relationships will be difficult to determine. Additional relativizations involving FewP are also presented here.

First, let us consider the question of whether or not there are sparse sets in FewP − P. Recall that this hypothesis has been shown to be equivalent to the existence of sparse sets in P that are not P-printable (Proposition 3.3) and to be a sufficient condition for the existence of a sparse set that is polynomial time Turing equivalent to a tally set and that is not self-P-printable (Theorem 3.6).

THEOREM 4.1. *There exists a sparse oracle S such that there are sparse sets in* $\mathrm{FewP}^S - P^S$.

Proof. Long ([16]) has shown that there exist sparse sets that are not polynomial time Turing equivalent to any tally set. These sets therefore are not self-P-printable. Let S be such a set. S is trivially in P^S. There is thus a sparse set that is in P^S but is not P^S-printable: S itself. By Proposition 3.3 this implies that there is a sparse set in $\mathrm{FewP}^S - \mathrm{P}^S$. □

Clearly, there are oracles relative to which there are no sparse sets in FewP − P, such as the oracles relative to which P = NP, but it is desirable to find one that is not quite so trivial. Specifically, we want an oracle relative to which P $\neq$ FewP and there are no sparse sets in FewP − P. Kurtz ([15]) constructed an oracle relative to which P $\neq$ NP and there are no sparse sets in NP − P. It turns out that the oracle he constructed in fact separates P from FewP as well.

THEOREM 4.2. *There exists a sparse oracle A such that $\mathrm{P}^A \neq \mathrm{FewP}^A$ and there are no sparse sets in $\mathrm{NP}^A - \mathrm{P}^A$.*

Proof. The proof of this theorem is not a construction, but a proof that the oracle constructed by Kurtz ([15]) which separates P and NP with no sparse sets in their difference also separates P and FewP. The actual construction will not be shown here, nor will the proof that relative to the oracle there are no sparse sets in NP − P. Only the assertion that the oracle separates P and FewP will be proven.

To begin with, it is necessary to understand some of the ideas and notation used in [15]. The following passages are from page 114 of that paper:

> Let L^A denote the language consisting of all strings σ for which there exists a τ of the same length such that $\sigma\tau \in A$. In notation, $L^A = \{\sigma : (\exists\tau)[|\sigma| = |\tau| \ \&\ \sigma\tau \in A]\}$.
>
> To ensure $\mathrm{NP} \neq \mathrm{P}$ we will construct A so that L^A is not in P^A. (It can be easily seen that L^A is in NP^A for all oracles A.) ...
>
> Our other goal is to ensure that there are no sparse sets in $\mathrm{NP}^A - \mathrm{P}^A$. To do this, we will attempt to code the sparse sets in NP^A into A so that they can be recovered in polynomial time. Let NP_e^A denote the eth NP^A language in some standard enumeration. Let p_e denote the eth polynomial in some standard enumeration of the polynomials. We can assume without loss of generality that NP_e^A is computable in nondeterministic time p_e; ...
>
> In order to describe the coding strategy, we first describe a tripling function with certain technically important properties. Each triple e, i, n will determine an odd integer $k_{e,i,n}$ unique to it. We will use the strings of length $k_{e,i,n}$ to code elements of NP_e^A of length n whenever there are fewer than $p_i(n)$ such elements. We may assume that $k_{e,i,n}$ is computable (in unary representation) in polynomial time from n (also in unary representation) for fixed e and i, and furthermore that $k_{e,i,n}$ is greater than $p_e(n)$, n, and $p_i(n)$. ...
>
> Now, if NP_e^A has fewer than $p_i(n)$ members, we will include the string $\sigma 0^{k_{e,i,n}-n}$ in A if and only if σ is in NP_e^A. Notice that because this coding string is of length $k_{e,i,n}$, it can be uniquely parsed. ...

By the definition of L^A, it can be seen that if A is sparse, then L^A is in FewP^A. It will be shown here that the oracle A constructed in [15] is sparse.

Let the notation $A^{=k_{e,i,n}}$ denote the set of strings in A of length exactly $k_{e,i,n}$. Only strings of length $k_{e,i,n}$ for some e, i, and n are added to A in the construction in [15]. By the above, if $A^{=k_{e,i,n}}$ is nonempty, then the number of strings in NP_e^A of length n is bounded by $p_i(n)$. This implies that $\|A^{=k_{e,i,n}}\| \leq p_i(n)$. But $k_{e,i,n}$ is greater than $p_i(n)$, so $\|A^{=k_{e,i,n}}\| < k_{e,i,n}$. This means that the number of strings in A of length m (substituting m for $k_{e,i,n}$) is less than m, so A is sparse. □

COROLLARY 4.3. *There exists a sparse oracle A such that $\mathrm{P}^A \neq \mathrm{FewP}^A$ and there are no sparse sets in $\mathrm{FewP}^A - \mathrm{P}^A$.*

Next it will be shown that there exist oracles for each of the four combinations of equality/inequality between P, FewP, and NP. First, the oracle constructed in [3] relative to which P = NP clearly makes P = FewP = NP. Next, the oracle constructed in [19] relative to which P ≠ UP = NP clearly makes P ≠ FewP = NP. The last two combinations require construction.

THEOREM 4.4. *There exists an oracle B such that $\mathrm{P}^B = \mathrm{FewP}^B \neq \mathrm{NP}^B$.*

Proof. This proof uses techniques similar to those found in [3], [19], and [13]. For any set A, define $L(A) = \{x \mid (\exists y \in A)(|y| = |x|)\}$. The function e is defined as $e(0) = 2$ and $e(n+1) = 2^{2^{e(n)}}$. B will be constructed so that $L(B) \in \mathrm{NP}^B - \mathrm{P}^B$ and $\mathrm{P}^B = \mathrm{FewP}^B$. P_i^A and NP_i^A will, respectively, represent the deterministic and nondeterministic ith oracle Turing machines in some standard enumeration with oracle A. They will also represent the languages of these machines, as determined by context. The running time of P_i^A and NP_i^A will be bounded by n^i. $B(n)$ will represent the set

of strings in B prior to stage n.

B will be constructed so that the following two requirements are met. First, for all i, $P_i^B \neq L(B)$, thus ensuring $L(B) \notin \mathrm{P}^B$. Note that for any set B, $L(B) \in \mathrm{NP}^B$, so this requirement actually ensures $L(B) \in \mathrm{NP}^B - \mathrm{P}^B$. The second requirement is that for all i and j, either there exists a string x such that $NP_i^B(x)$ has greater than $|x|^j$ accepting computations (i.e., $NP_i^B \notin \mathrm{FewP}^B$) or $NP_i^B \in \mathrm{P}^B$. This ensures that $\mathrm{P}^B = \mathrm{FewP}^B$.

This will be accomplished by considering two types of conditions and what it means for them to be fulfillable, and then fulfilling some of them. The first type of condition, *condition* i, is considered to be *fulfillable at stage* n if $e(n)^i < 2^{e(n)}$. The second type of condition, *condition* $\langle i, j \rangle$, is considered to be *fulfillable at stage* n if there exists a string x and a set D of strings of length $e(n)$ such that $NP_i^{B(n) \cup D}(x)$ has more than $|x|^j$ accepting computations and $|x| < e(n+1)$. We assume a total ordering on these conditions, such as letting the number $0i$ represent condition i and letting the number $1\langle i, j \rangle$ represent condition $\langle i, j \rangle$. Initially, all conditions are marked unfulfilled.

The construction is as follows. At stage 0, let $B(0) = A$, where A is a PSPACE-complete set containing only strings of odd lengths. We will add only even length strings (specifically, only strings of length $e(n)$ will be added at stage n), so B can be considered to be the disjoint union of two sets: A and the sets of strings added later.

At stage n, find the least fulfillable condition not yet fulfilled. If no such condition exists, go to stage $n+1$. Otherwise, if this condition is condition i, run $P_i^{B(n)}$ on $0^{e(n)}$. If this accepts, simply mark condition i fulfilled. If this rejects, put the least string of length $e(n)$ not queried in the computation of $P_i^{B(n)}(0^{e(n)})$ into B and mark condition i fulfilled. Because there are at most $e(n)^i$ queries in this computation (there are at most that many steps) and $e(n)^i < 2^{e(n)}$ (by the definition of fulfillable), we know that such a string exists.

If the least unfulfilled fulfillable condition is condition $\langle i, j \rangle$, add the elements of the set D (from the definition of the condition being fulfillable) to B and mark condition $\langle i, j \rangle$ fulfilled.

There are a number of observations to be made about the oracle B thus constructed. When fulfilling condition i, we ensure that $P_i^B \neq L(B)$ as follows. If $P_i^{B(n)}(0^{e(n)})$ accepts, then no strings of length $e(n)$ are added to B, so $0^{e(n)} \notin L(B)$. If $P_i^{B(n)}(0^{e(n)})$ rejects, some string of length $e(n)$ not affecting this computation is put into B, so all strings of length $e(n)$, including $0^{e(n)}$, are in $L(B)$. Strings added to B at stage $n+1$ and later are of length $e(n+1) = 2^{2^{e(n)}}$ and greater, so they cannot be queried by $P_i^{B(n)}$ and therefore cannot affect the computation.

For every i there exists an n such that for all $n' > n$, $e(n')^i < 2^{e(n')}$, so every condition i becomes fulfillable and remains so until it is eventually the least unfulfilled fulfillable condition, at which time it is fulfilled. Thus, for every i, $P_i^B \neq L(B)$, so $L(B) \notin \mathrm{P}^B$.

Now consider NP_i^B. In fulfilling condition $\langle i, j \rangle$, by adding D to B we are attempting to keep NP_i^B out of FewP^B. If for all j, condition $\langle i, j \rangle$ is fulfilled, then we have succeeded in this.

It will now be shown that if this is not the case, i.e., if $NP_i^B \in \mathrm{FewP}^B$, then $NP_i^B \in \mathrm{P}^B$, so $\mathrm{P}^B = \mathrm{FewP}^B$. If condition $\langle i, j \rangle$ is not fulfilled, then for all $j' > j$, condition $\langle i, j' \rangle$ is also not fulfilled, so let j be the least j such that condition $\langle i, j \rangle$ is unfulfilled.

Membership in NP_i^B can be determined using the following algorithm by a P^B machine. On input x, let n be the unique number such that $e(n-1) \le \log|x| < e(n)$. For sufficiently long x, $|x|^i < e(n+1)$ and condition $\langle i,j\rangle$ is not fulfillable at stage n, so we assume both of these to be true. Because all strings in $B(n) - A$ have length less than or equal to $e(n-1) \le \log|x|$, $2|x|$ queries to B (one for each string with length less than or equal to $\log|x|$) is sufficient to deterministically construct $B(n) - A$. Strings added at stage n and later have length greater than or equal to $e(n) \ge \log|x|$, so they cannot be inadvertently added in the construction of $B(n) - A$.

If (W, W') are disjoint sets of strings of length $e(n)$, $W \subseteq B$ and $W' \subseteq \overline{B}$, define an *accepting computation of NP_i with respect to (W, W')* as an accepting computation by NP_i where a query y is answered as follows. If $|y| \le e(n-1)$, then answer "yes" if $y \in B$ (i.e., $y \in B(n)$) and "no" if $y \in \overline{B}$ (i.e., $y \in \overline{B(n)}$). If $|y| = e(n)$, then answer "yes" if $y \in W$, "no" if $y \in W'$, and consistently with the other queries in the computation if y is in neither W nor W'. Because $|x|^i < e(n+1)$, we know that queries of length greater than $e(n)$ cannot be in $B(n)$, so they will always be answered "no." Queries of odd length are answered according to their membership in A.

The queries in neither W nor W' and their answers are stored in a table called the *core* of the accepting computation. If S is the core of an accepting computation, S^{yes} and S^{no} will denote those queries answered "yes" and "no," respectively, that appear in the core. The same symbol will sometimes be used to denote both an accepting computation and its core.

LEMMA 4.5. *Let (W, W') be disjoint sets of strings of length $e(n)$, $W \subseteq B$ and $W' \subseteq \overline{B}$, and condition $\langle i,j\rangle$ not fulfillable at stage n. If $S_1, S_2, \cdots, S_{|x|^j+1}$ are the cores of $|x|^j + 1$ different accepting computations of $NP_i(x)$ with respect to (W, W'), then there exists k_1 and k_2, $1 \le k_1 < k_2 \le |x|^j + 1$ such that $((S_{k_1}^{yes} \cap S_{k_2}^{no}) \cup (S_{k_1}^{no} \cap S_{k_2}^{yes})) \ne \emptyset$ (i.e., they have at least one query in common with different answers).*

Proof. If this is not the case, then $D = (\bigcup_{k=1}^{|x|^j+1} S_k^{yes} \cup W)$ would fulfill condition $\langle i,j\rangle$ at stage n, which is a contradiction. □

Continuing now with the algorithm, initially let $W = W' = \emptyset$. Repeat the following loop until x is either accepted or rejected, or until $|x|^i$ iterations of the loop have been made. Guess a set of $|x|^j$ accepting computations of $NP_i(x)$ with respect to (W, W') that are consistent with each other, if it exists. If there are fewer than $|x|^j$ consistent accepting computations, guess instead a set with the greatest number of mutually consistent accepting computations. This can be done deterministically using oracle B because it includes a disjoint PSPACE-complete oracle. If no such accepting computation exists, reject. For each accepting computation in the set, check the strings in its core with the oracle B. If one or more of the cores agree with B, accept; otherwise update W and W' by putting the queries with "yes" answers into W and the queries with "no" answers into W' and repeat the loop.

If $x \in NP_i^B$, let S be an accepting computation. Each iteration of the loop determines at least one query in S in one of two possible ways. If $|x|^j$ consistent computations are found, because S is also an accepting computation of $NP_i(x)$ with respect to (W, W'), S along with the guessed computations makes $|x|^j + 1$ accepting computations, so the above lemma applies. Because the guessed computations are mutually consistent, the disagreement must be with S. If fewer mutually consistent accepting computations are found, there must be some disagreement between the query answers of one of the cores in the largest set of mutually accepting computations and S; otherwise S would be included in this set, in which case *all* the queries in S are determined. Because S contains at most $|x|^i$ queries, that is the maximum number

of iterations of the loop required to determine an accepting computation of $NP_i(x)$ with respect to (W, W') that agrees with B, so x can be accepted.

If no such accepting computation is found within $|x|^i$ iterations, then there is no accepting computation S, so x is rejected. □

Finally, using techniques based upon those in [7], where an oracle was constructed to separate UP from both P and NP, an oracle will be constructed relative to which not only is FewP separated from both P and NP, but relative to which P $\neq$ UP $\neq$ FewP $\neq$ NP.[1]

Let us first describe a simple board covering problem, a variation on that in [7] (where it is called a pebbling game). An $m \times m$ *board* is an $m \times m$ matrix over $\{0, 1\}$ with its m^2 squares denoted b_{ij}, where $1 \leq i, j \leq m$. A square b_{ij} is *pebbled* (i.e., there is a pebble on that square) if and only if $b_{ij} = 1$. An $m \times m$ board is said to be *covered* if for all i and j, $1 \leq i, j \leq m$, $b_{ii} = 1$ and either $b_{ij} = 1$ or $b_{ji} = 1$ (or both). To cover an $m \times m$ board clearly requires at least $m^2/2$ pebbles, the number to cover half the squares. This is as described in [7].

We extend this with the following. For any k, an $m \times m$ board is said to be *k-weakly-covered* if $b_{ii} = 1$ for all i inclusively between 1 and m and if for every set $I \subseteq \{i \mid 1 \leq i \leq m\}$ with cardinality greater than $(\log m)^k$, there exist distinct $i, j \in I$ such that $b_{ij} = 1$ or $b_{ji} = 1$ (or both). For all $m > 1$, 0-weakly-covering an $m \times m$ board is equivalent to covering it. If a board is k-weakly-covered, it is also $(k+1)$-weakly-covered.

LEMMA 4.6. *To k-weakly-cover an* $m \times m$ *board requires at least* $m(m-1)/z(z-1)$ *pebbles, where* $z = \lfloor (\log m)^k + 1 \rfloor$.

Proof. To k-weakly-cover an $m \times m$ board, it is necessary that for every $I \subseteq \{i \mid 1 \leq i \leq m\}$ with cardinality $z = \lfloor (\log m)^k + 1 \rfloor$ there exist distinct $i, j \in I$ such that $b_{ij} = 1$ or $b_{ji} = 1$. If this property holds for I, we say that I is satisfied. We say that a pair i, j is satisfied if $b_{ij} = 1$ or $b_{ji} = 1$. In this proof only I's of cardinality z will be considered.

So at least $\binom{m}{z}$ many I's must be satisfied. A given pair i, j occurs in $\binom{m-2}{z-2}$ many I's, the number of ways the remaining $z-2$ elements of I can be chosen from the remaining $m-2$ numbers between 1 and m after fixing i and j. Thus, satisfying a pair i, j satisfies at most $\binom{m-2}{z-2}$ I's, so

$$\binom{m}{z} \Big/ \binom{m-2}{z-2} = \frac{m(m-1)}{z(z-1)}$$

pairs must be satisfied, requiring that many pebbles. □

This lower bound is not intended to be tight, and does not even take into account the fact that the diagonal must be pebbled. It is, however, sufficient for its use in the following theorem.

THEOREM 4.7. *There exists a recursive oracle* C *such that* $\mathrm{P}^C \neq \mathrm{UP}^C \neq \mathrm{FewP}^C \neq \mathrm{NP}^C$.

Proof. The proof begins with the following definition.

DEFINITION 4.1. For any oracle X, we define the following three sets:

$L_0(X) = \{0^n \mid$ there exists a $y \in X$ such that $|y| = n$ and n modulo 3 is $0\}$,
$L_1(X) = \{0^n \mid$ there exists a $y \in X$ such that $|y| = n$ and n modulo 3 is $1\}$,
$L_2(X) = \{0^n \mid$ there exists a $y \in X$ such that $|y| = n$ and n modulo 3 is $2\}$.

[1] Since this result was first presented in [20], Beigel ([5]) independently constructed an oracle that provides finer separation of classes based on the number of accepting computations.

We will say that a number is "m_0" ("m_1", "m_2") if modulo 3 it is equal to 0 (1, 2, respectively).

For every oracle C with at most one string per m_0 length and at most two strings per m_1 length, $L_0(C) \in \mathrm{UP}^C$, $L_1(C) \in \mathrm{FewP}^C$, and $L_2(C) \in \mathrm{NP}^C$, so it suffices to construct C as such, with the additional requirements that $L_0(C) \notin \mathrm{P}^C$, $L_1(C) \notin \mathrm{UP}^C$, and $L_2(C) \notin \mathrm{FewP}^C$.

The construction will be done using three types of stages: stage s, stage s', and stage s''. At stage s we will ensure that $P_i^C \neq L_0(C)$, where $i = s$, adding only m_0 length strings, at most one per length. Doing this for every s tells us that $\mathrm{P}^C \neq \mathrm{UP}^C$.

At stage s' we will ensure that either $NP_i^C \neq L_1(C)$ or that NP_i^C accepts some string with more than one accepting computation, where $i = s$. This will be accomplished while adding only m_1 length strings, at most two per length. Doing this for every s' tells us that $\mathrm{UP}^C \neq \mathrm{FewP}^C$.

At stage s'', where $s = \langle i, j\rangle$, we will ensure that either $NP_i^C \neq L_1(C)$ or that NP_i^C accepts some string x with greater than $|x|^j$ accepting computations. This will be accomplished while adding only m_2 length strings. For a given i, fulfilling the second alternative for every j tells us that $NP_i^C \notin \mathrm{FewP}^C$. Because there must be some i such that $NP_i^C = L_1(C)$, satisfying every s'' tells us that $\mathrm{FewP}^C \neq \mathrm{NP}^C$.

The order in the construction will be stage 1, stage $1'$, stage $1''$, stage 2, stage $2'$, stage $2''$, etc. Initially, $m = 0$ and $C = \emptyset$.

At stage s, choose an m_0 integer n large enough that $n > m$ and $n^i < 2^n$, where $i = s$. Run $P_i^{C(s)}$ on 0^n. If $P_i^{C(s)}(0^n)$ accepts, set $m = 2^n$ and continue to the next stage. If $P_i^{C(s)}(0^n)$ rejects, add to C the least string of length n not queried during the computation. Because there are at most n^i queries (there are at most that many steps) and $n^i < 2^n$, we know that such a string exists. Set $m = 2^n$ and continue to the next stage.

At stage s', choose an m_1 integer n large enough that $n > m$ and $n^i < 2^{n-1}$, where $i = s$. If there is some string x of length less than or equal to n such that $NP_i^{C(s')}(x)$ has more than one accepting computation, add nothing to C, set $m = 2^n$, and continue to the next stage.

If for every x of length less than or equal to n, $NP_i^{C(s')}(x)$ has at most one accepting computation, then run $NP_i^{C(s')}$ on 0^n. If this accepts, add nothing to C, set $m = 2^n$, and continue to the next stage. If $NP_i^{C(s')}(0^n)$ rejects, add a nonempty set X of strings of length n to C such that $NP_i^{C(s')\cup X}$ either rejects 0^n or accepts 0^n ambiguously, and $\|X\| = 1$ or $\|X\| = 2$. (Lemma 4.8 will show that such an X exists.) Then set $m = 2^n$ and continue to the next stage.

At stage s'', where $s = \langle i, j\rangle$, choose an m_2 integer n large enough that $n > m$, $n^{i+j} < 2^{n/4}$, and $(n^j + 1)(n^j) < 2^{n/4}$. If there is some string x of length less than or equal to n such that $NP_i^{C(s'')}(x)$ has greater than $|x|^j$ accepting computations, add nothing to C, set $m = 2^n$, and continue to the next stage.

If for every x of length less than or equal to n, $NP_i^{C(s'')}(x)$ has at most $|x|^j$ accepting computations, run $NP_i^{C(s'')}$ on 0^n. If this accepts, add nothing to C, set $m = 2^n$, and continue to the next stage. If $NP_i^{C(s'')}(0^n)$ rejects, add a set X of strings of length n to C such that $NP_i^{C(s'')\cup X}(0^n)$ either rejects or accepts with greater than n^j accepting computations. (Lemma 4.9 will show that such an X exists.) Then set $m = 2^n$ and continue to the next stage.

LEMMA 4.8. *In stage s', where $i = s$, at the point where $NP_i^{C(s')}(0^n)$ rejects,*

there exists a set X of strings of length n such that $NP_i^{C(s')\cup X}(0^n)$ either rejects or accepts ambiguously, and $\|X\| = 1$ or $\|X\| = 2$.

Proof. The set X can be constructed as follows. For each string x of length n, determine the number of accepting computations of $NP_i^{C(s'')\cup\{x\}}(0^n)$. If the number is either 0 (i.e., it rejects) or is greater than one, $X = \{x\}$.

Otherwise, we have a situation where $NP_i^{C(s')}(0^n)$ rejects, but for each string x of length n, $NP_i^{C(s'')\cup\{x\}}(0^n)$ accepts unambiguously. An accepting path of $NP_i^{C(s')\cup\{x\}}(0^n)$ is called a *critical path for x*. Over all the x's of length n, there are then a total of at most 2^n critical paths. Denote $cr(x)$ as the set of queries of length n in the critical path for x. For each x, $x \in cr(x)$ because $NP_i^{C(s')}(0^n)$ rejects, while $NP_i^{C(s')\cup\{x\}}(0^n)$ accepts.

Adding a string y of length n to the oracle cannot affect the computation of the critical path for x if $y \notin cr(x)$. Thus, if we can find a pair of strings of length n such that neither is in the critical path of the other, then the set of these two strings qualifies as X so that there is more than one accepting computation of $NP_i^{C(s')\cup X}(0^n)$.

To show that such a set exists, the problem of finding the strings of length n that do not appear in each other's critical paths can be modeled with the aforementioned board covering problem. The board has dimensions $2^n \times 2^n$, where each dimension represents the set of strings of length n in their standard lexicographical ordering. For any pair of strings x_1, x_2, each of length n, $b_{x_1 x_2} = 1 \iff x_2 \in cr(x_1)$.

The total number of critical paths is at most 2^n, each with at most n^i queries, so the total number of queries in the critical paths is at most $n^i 2^n < 2^{n-1}2^n = 2^{2n-1}$ (because we chose n large enough that $n^i < 2^{n-1}$). The minimum number of pebbles needed to cover a $2^n \times 2^n$ board is $2^{2n}/2 = 2^{2n-1}$, so there are not enough queries to cover the board.

Because the $2^n \times 2^n$ board cannot be covered, it must be that either for some x, $b_{xx} = 0$ (which is not the case in this setting) or there exist x_1 and x_2 such that $b_{x_1 x_2} = b_{x_2 x_1} = 0$. This implies that $\{x_1, x_2\}$ qualifies as X so that there is more than one accepting computation of $NP_i^{C(s')\cup X}(0^n)$. (The description for this stage, with the exception of the restriction of the oracle to have at most two elements added, is contained in different words in [7].) □

LEMMA 4.9. *In stage s'', where $s = \langle i, j\rangle$, at the point where $NP_i^{C(s'')}(0^n)$ rejects, there exists a nonempty set X of strings of length n such that $NP_i^{C(s'')\cup X}(0^n)$ either rejects or accepts with greater than n^j accepting computations.*

Proof. The set X can be constructed as follows. For each string x of length n, determine the number of accepting computations of $NP_i^{C(s'')\cup\{x\}}(0^n)$. If the number is either 0 (i.e., it rejects) or is greater than n^j, $X = \{x\}$.

Otherwise, we have a situation where $NP_i^{C(s'')}(0^n)$ rejects, but for each string x of length n, $NP_i^{C(s'')\cup\{x\}}(0^n)$ accepts with at most n^j accepting computations. As before, an accepting path of $NP_i^{C(s'')\cup\{x\}}(0^n)$ is called a *critical path for x*. Over all the x's of length n there are then a total of at most $n^j 2^n$ critical paths. Let $cr(x)$ denote the set of queries of length n in all the critical paths for x, and again for each x, $x \in cr(x)$ because $NP_i^{C(s'')}(0^n)$ rejects while $NP_i^{C(s'')\cup\{x\}}(0^n)$ accepts.

Adding a string y of length n to the oracle cannot affect the computation along any critical path for x if $y \notin cr(x)$. Thus, if we can find greater than n^j strings of length n such that none of them is in the critical path of another, then the set of

these strings qualifies as X so that there are greater than n^j accepting computations of $NP_i^{C(s'')\cup X}(0^n)$.

As in stage s', we show that such a set exists by using the board covering problem to model the problem of finding the strings of length n that do not appear in each other's critical paths. The board has dimensions $2^n \times 2^n$, where each dimension represents the set of strings of length n in their standard lexicographical ordering. For any pair of strings x_1, x_2, each of length n, $b_{x_1x_2} = 1 \iff x_2 \in cr(x_1)$.

The total number of critical paths is at most $n^j 2^n$, each with at most n^i queries, so the total number of queries in the critical paths is at most $n^{i+j}2^n < 2^{n/4}2^n = 2^{(5/4)n}$ (because we chose n large enough that $n^{i+j} < 2^{n/4}$). The minimum number of pebbles needed to j-weakly-cover a $2^n \times 2^n$ board is

$$\frac{(2^n)(2^n-1)}{(n^j+1)(n^j)} > \frac{(2^n)(2^n-1)}{2^{n/4}} = \frac{2^{2n}-2^n}{2^{n/4}} > \frac{2^{2n-1}}{2^{n/4}} = 2^{(7/4)n-1} > 2^{(5/4)n}.$$

Thus there are not enough queries to j-weakly-cover the board.

Because the $2^n \times 2^n$ board cannot be j-weakly-covered, it must be that either there is some $x \in \{0,1\}^n$ for which $b_{xx} = 0$ (which is not the case in this setting), or there exists a set $I \subseteq \{0,1\}^n$ with cardinality greater than $(\log 2^n)^j = n^j$ such that for all distinct $x_1, x_2 \in I$, $b_{x_1x_2} = b_{x_2x_1} = 0$. This implies that the set I qualifies as X so that there are greater than n^j accepting computations of $NP_i^{C(s'')\cup X}(0^n)$. □

The oracle simultaneously separating P, UP, FewP, and NP is thus constructed. □

There are still four other combinations of equality/inequality regarding the relationships between P, UP, FewP, and NP for which oracles have not been been constructed here (or at least it has not been proven that the constructed oracles possess the desired property, though some of them must have been constructed). These do not appear to come directly from the techniques used here, and require additional consideration.

5. Acknowledgments. I would like to thank the circa 1988 Northeastern University/Boston University theory group, especially my thesis advisor Alan Selman, for support during early versions of these results. Thanks also to the anonymous referees for their assistance.

REFERENCES

[1] E. ALLENDER, *The complexity of sparse sets in* P, in Proc. Conference on Structure in Complexity Theory, A. Selman, ed., Springer-Verlag, Berlin, New York, 1986, pp. 1–11.

[2] E. ALLENDER AND R. RUBINSTEIN, P-*printable sets*, SIAM J. Comput., 17 (1988), pp. 1193–1202.

[3] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the* P =? NP *question*, SIAM J. Comput., 4 (1975), pp. 431–441.

[4] J. BALCÁZAR AND R. BOOK, *Sets with small generalized Kolmogorov complexity*, Acta Informatica, 23 (1986), pp. 679–688.

[5] R. BEIGEL, *On the relativized power of additional accepting paths*, in Proc. IEEE Structure in Complexity Theory Fourth Annual Conference, Eugene, OR, 1989, pp. 216–224.

[6] L. BERMAN, *Polynomial reducibilities and complete sets*, Ph.D. thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, 1977.

[7] J. GESKE AND J. GROLLMANN, *Relativizations of unambiguous and random polynomial time classes*, SIAM J. Comput., 15 (1986), pp. 511–519.

[8] J. GROLLMANN AND A. SELMAN, *Complexity measures for public-key cryptosystems*, in Proc. 25th IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984, pp. 495–503.

[9] J. HARTMANIS, *Generalized Kolmogorov complexity and the structure of feasible computations*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 439–445.

[10] J. HARTMANIS AND L. HEMACHANDRA, *On sparse oracles separating feasible complexity classes*, in Proc. 3rd Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, Berlin, New York, 1986, pp. 321–333.

[11] J. HARTMANIS, N. IMMERMAN, AND V. SEWELSON, *Sparse sets in* NP-P*: EXPTIME versus* NEXPTIME, Inform. and Control, 65 (1985), pp. 158–181.

[12] J. HARTMANIS AND Y. YESHA, *Computation times of* NP *sets of different densities*, Theoret. Comput. Sci., 34 (1984), pp. 17–32.

[13] S. HOMER AND A. SELMAN, *Oracles for structural properties: The isomorphism problem and public-key cryptography*, in Proc. IEEE Structure in Complexity Theory Fourth Annual Conference, Eugene, OR, 1989, pp. 3–14.

[14] K. KO, *Continuous optimization problems and a polynomial hierarchy of real functions*, J. Complexity, 1 (1985), pp. 210–231.

[15] S. KURTZ, *Sparse sets in* NP-P*: Relativizations*, SIAM J. Comput., 14 (1985), pp. 113–119.

[16] T. LONG, *On restricting the size of oracles compared with restricting access to oracles*, SIAM J. Comput., 14 (1985), pp. 585–597.

[17] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618–627.

[18] S. MORAN, *On the accepting density hierarchy in* NP, SIAM J. Comput., 11 (1982), pp. 344–349.

[19] C. RACKOFF, *Relativized questions involving probabilistic algorithms*, J. Assoc. Comput. Mach., 29 (1982), pp. 261–268.

[20] R. RUBINSTEIN, *Structural complexity classes of sparse sets: Intractability, data compression and printability*, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, MA, 1988.

[21] U. SCHÖNING, *Complexity and Structure*, Lecture Notes in Computer Science 211, Springer-Verlag, Berlin, New York, 1986.

[22] L. VALIANT, *Relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1034–1045, December 1991
© 1991 Society for Industrial and Applied Mathematics
004

NEW UPPER BOUNDS IN KLEE'S MEASURE PROBLEM*

MARK H. OVERMARS† AND CHEE-KENG YAP‡

Abstract. New upper bounds for the measure problem of Klee are given which significantly improve the previous bounds for dimensions greater than two. An $O(n^{d/2}\log n, n)$ time-space upper bound is obtained and used to compute the measure of a set of n boxes in Euclidean d-space. The solution is based on a new data structure, which is called an orthogonal partition tree. This structure has other applications as well.

Key words. measure, partition trees, computational geometry

AMS(MOS) subject classifications. 68U05, 68Q25

1. Introduction. In 1977, Klee [5] posed the *measure problem*: given a set of n intervals (of the real line), find the length of their union. He gave an $O(n \log n)$ time solution and asked if this was optimal. This generated considerable interest in the problem, and shortly after, Fredman and Weide [4] proved that $\Omega(n \log n)$ is a lower bound under the usual model of computation. Bentley [2] considered the natural extension to d-dimensional space where we ask for the d-dimensional measure of a set of d-rectangles. He showed that the $O(n \log n)$ bound holds for $d = 2$ as well, and, for $d > 2$, the result generalizes to an upper bound of $O(n^{d-1} \log n)$. Thus the results are optimal for $d = 1, 2$. We refer to [7] for an account. Concerning these results for $d \geq 3$, Preparata and Shamos remarked in their book ([7, pp. 328–329]):

> What is grossly unsatisfactory about the outlined method for $d \geq 3$ is the fact that there is a "coherence" between two consecutive sections in the sweep that we are unable to exploit.... Although it seems rather difficult to improve on this result, no conjecture about its optimality has been formulated.

The only progress made since this account was published in 1985 was a small improvement by van Leeuwen and Wood [8], who removed the $\log n$ factor from Bentley's upper bound for $d \geq 3$. The test case seems to be $d = 3$: is $O(n^2)$ really necessary for computing the volume of a set of n boxes in 3-space? In this paper we show that $O(n^{1.5} \log n)$ suffices.

The idea is to use a plane-sweep approach and dynamically maintain the measure of a set of two-dimensional rectangles in time $O(\sqrt{n} \log n)$ per update. Such a result means that we can maintain the area of a set of rectangles *implicitly* without having to represent the full boundary structure. This is because any explicit representation of the boundary of n rectangles requires $\Omega(n^2)$ time in the worst case because of the simple "trellis" example (see Fig. 1): it consists of n long vertical rectangles which are pairwise disjoint, superposed on n long horizontal rectangles, also disjoint among themselves.

The first idea is to exploit the regularity of such trellis structures by maintaining only $O(n)$ amount of information (at the boundary of the box containing the trellis) to keep track of the area of the trellis rectangles. Of course, a union of rectangles is too irregular to be consistently exploited in this way, so the next idea is to partition the

* Received by the editors February 20, 1990; accepted for publication February 1, 1991.

† Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. The research of this author was partially supported by the ESPRIT II Basic Research Actions Program of the European Community under contract 3075 (project ALCOM).

‡ Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012. The research of this author has been supported by Office of Naval Research grant N00014-85-K-0046, and National Science Foundation grants DCR-84-01898 and CCR-87-03458.

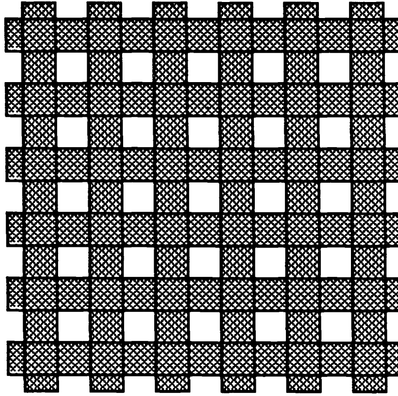


FIG. 1. *Trellis.*

plane into a collection of trellises. Using a generalization of the k-d tree of Bentley [1], we are able to form such a partition with only $O(n)$ trellises, each of size $O(\sqrt{n})$. As we shall see, extending this to higher dimensions requires a partition with interesting properties that might be useful for other applications as well.

The rest of this paper is organized as follows: §2 describes the basic space sweep algorithm we use and introduces the generalized k-d tree, which we will call an orthogonal partition tree, for storing the boxes. Section 3 contains the solution to the three-dimensional measure problem. Section 4 generalizes this solution to a d-dimensional method, using an interesting partition scheme of the d-dimensional space. This results in an $O(n^{d/2} \log n, n^{d/2})$ time-space upper bound. In §5 we exploit a streaming technique of Edelsbrunner and Overmars [3] to reduce the amount of storage required to $O(n)$ only, for any dimension d. In §6 we briefly mention some other applications of the method, e.g., computing the measure of the boundary of the union of a set of boxes. Finally, in §7, some conclusions, extensions, and open problems are given.

Throughout the paper we will use the following terminology. A *d-box* is the cartesian product of d intervals in d-dimensional space. The *i-boundaries* of a d-box are the parts of the boundary that are perpendicular to the ith coordinate axis. Each d-box has two i-boundaries for $1 \leq i \leq d$. We refer to them as the left and right i-boundary. The *i-interval* is the projection of the d-box on the x_i-axis. For a d-box R, we denote with $\mathrm{Int}(R)$ the interior of R.

DEFINITION 1.1. A d-box R_1 is said to *partially cover* R_2 if the boundary of R_1 intersects $\mathrm{Int}(R_2)$. R_1 is said to (*completely*) *cover* R_2 if $R_2 \subseteq R_1$.

DEFINITION 1.2. For two d-boxes R_1 and R_2 we say that R_1 is an *i-pile* with respect to R_2 if R_1 partially covers R_2 and, for all $1 \leq j \leq d$ with $j \neq i$, the j-interval of R_2 is fully contained in the j-interval of R_1.

In other words, in each direction, except for direction i, R_1 completely covers R_2. i-piles will play an important role in this paper.

An extended abstract of this paper appeared in [6].

2. General framework. The basic method for solving the d-dimensional measure problem is as follows. Let V be the set of n d-boxes for which we want to compute the measure. Let $V' = \{a_1, \cdots, a_{n'}\}$ be the set of all different x_d-coordinates of vertices of the boxes, i.e., all different endpoints of d-intervals. We sort the boxes both by left and right d-boundary. We solve the measure problem using a space-sweep approach turning the static d-dimensional problem into a dynamic $(d-1)$-dimensional problem. We sweep a hyperplane along the dth coordinate axis stopping at each value in V'. During the sweep we maintain the $(d-1)$-dimensional measure of the boxes intersected by the sweep plane. At each step of the sweep we multiply this $(d-1)$-dimensional measure by the distance traveled with the sweep hyperplane and add this to the measure found so far. The algorithm looks as follows.

S:=$\emptyset$;
$MEAS$:=0;
for i:=1 **to** $n'-1$ **do**
  Insert all d-boundaries of boxes that start at a_i in S;
  M:=$(d-1)$-dimensional measure of boxes in S;
  $MEAS$:=$MEAS + (a_{i+1} - a_i) \times M$;
  Delete all d-boundaries of boxes that end at a_{i+1} from S
end;

At termination, $MEAS$ will contain the measure of the set of boxes. S will be a dynamic data structure for maintaining the $(d-1)$-dimensional measure. If insertions and deletions in S can be performed in time $F_{d-1}(n)$, the method will take time $O(n \log n + nF_{d-1}(n))$. This approach is due to Bentley.

To maintain the measure of the set of boxes intersected by the sweep-plane, we introduce a generalization of the k-d tree.

DEFINITION 2.1. A d-dimensional orthogonal partition tree is a balanced binary tree. With each internal node δ is associated a region C_δ of the d-dimensional space, with the following properties:

- C_{root} is the whole d-dimensional space.
- For each node δ, C_δ is a (possibly unbounded) d-box.
- For each node δ, with sons δ_1 and δ_2, $Int(C_{\delta_1}) \cap Int(C_{\delta_2}) = \emptyset$ and $C_{\delta_1} \cup C_{\delta_2} = C_\delta$.

C_δ will be called the *region associated with* δ. When δ is a leaf we refer to C_δ as a *cell*. It immediately follows that for each full level of the orthogonal partition tree all regions are essentially disjoint and their union is the d-dimensional space. From now on we drop the qualifying word "orthogonal" and speak only of "partition trees," which are not to be confused with the "nonorthogonal" partition trees of, Willard [11] and Welzl [10], for example.

To use partition trees for maintaining the measure of a set of d-boxes we store the following extra information in the partition tree: With each leaf δ we store all boxes that intersect $\mathrm{Int}(C_\delta)$ but do not cover the region associated with the father of δ. For each internal node δ we store a counter TOT_δ that contains the number of d-boxes that completely cover C_δ but only partially cover $C_{father(\delta)}$. Finally, with each node δ we associate a field M that is defined as follows: If δ is a leaf, M contains the measure of the boxes stored at δ restricted to C_δ. Otherwise, if $TOT_\delta > 0$ then M is the measure of C_δ; otherwise $M = M_{lson(\delta)} + M_{rson(\delta)}$. It is easy to verify that M_{root} is the measure of the set of d-boxes.

To maintain the measure in a dynamically changing set, we have to be able to insert and delete d-boxes in the partition tree. The basic insertion algorithm is the following:

```
procedure Insert(box,δ);
    if δ is a leaf then
        Store box at δ;
        Recompute M_δ
    else if box covers C_δ then
        TOT_δ:=TOT_δ + 1;
        M_δ:=measure of C_δ
    else if box partially covers C_δ then
        Insert(box,lson(δ));
        Insert(box,rson(δ));
        if TOT_δ > 0 then M_δ:= measure of C_δ else M_δ:=M_lson(δ) + M_rson(δ) end
    end;
```

The routine is invoked via Insert(*box*,*root*). The deletion routine is similar:

```
procedure Delete(box,δ);
    if δ is a leaf then
        Remove box at δ;
        Recompute M_δ
    else if box covers C_δ then
        TOT_δ:=TOT_δ - 1;
        if TOT_δ > 0 then M_δ:= measure of C_δ else M_δ:=M_lson(δ) + M_rson(δ) end
    else if box partially covers C_δ then
        Delete(box,lson(δ));
        Delete(box,rson(δ));
        if TOT_δ > 0 then M_δ:= measure of C_δ else M_δ:=M_lson(δ) + M_rson(δ) end
    end;
```

The routine is invoked via Delete(*box*,*root*). Note the similarity with the methods of Bentley [2] and van Leeuwen and Wood [8] for the one- and two-dimensional cases. The main difference is that we no longer insist that the leaves be fully covered by the boxes that intersect them. It is immediately clear that the amount of time required depends on the number of nodes visited and the amount of time required for computing the measure at the leaves. In the sequel of this paper we will show that partition trees exist in which both are small.

3. Dynamic measure problem in two dimensions. To illustrate the general solution that we will develop in the next section, we first solve the three-dimensional measure problem. Solving the three-dimensional problem means that we have to design a two-dimensional partition tree with good performance. To obtain such a partition tree, we first define a subdivision of the plane into rectangular cells with some interesting properties.

Let V be the set of the rectangles that will be inserted and deleted in the partition tree. First we split the x_1-axis into $2\sqrt{n}$ intervals such that each interval contains less than or equal to $\sqrt{n}$ 1-boundaries of rectangles. This defines $2\sqrt{n}$ slabs in the plane. Each slab s will be split by horizontal line segments into a number of cells. Let V_s^1 be the set of rectangles that have a 1-boundary inside s. Let V_s^2 be the set of rectangles that only have a 2-boundary intersecting s. (Note that the size of V_s^1 is bounded by $\sqrt{n}$, but the size of V_s^2 can be almost n.) We draw a line segment along each 2-boundary of a rectangle in V_s^1. Moreover, we draw a line segment along each $\sqrt{n}$-th 2-boundary of a rectangle in V_s^2. In this way s is partitioned into $\leq 4\sqrt{n}$ rectangular cells.

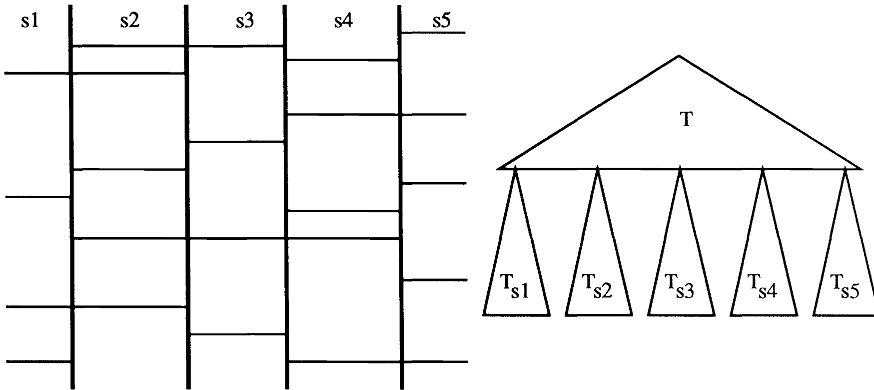


FIG. 2. *The two-dimensional partition tree.*

LEMMA 3.1. *The partition has the following properties:*

1. *There are $O(n)$ cells.*
2. *Each rectangle of V partially covers at most $O(\sqrt{n})$ cells.*
3. *No cell contains vertices in its interior.*
4. *Each cell has at most $O(\sqrt{n})$ rectangles partially covering it.*

Proof. Property 1 follows from the fact that there are $2\sqrt{n}$ slabs, each with $4\sqrt{n}$ cells.

To show property 2, note that each vertical line cuts through one slab, i.e., through at most $4\sqrt{n}$ cells. Each horizontal line cuts in each slab through one cell, hence, through at most $2\sqrt{n}$ cells in total. V partially covers a cell only if its boundary cuts through the cell. This boundary consists of two vertical and two horizontal line segments. Hence, it cuts through at most $12\sqrt{n}$ cells.

Property 3 follows from the fact that, if a vertex lies in the interior of a slab s, it belongs to a rectangle in V_s^1 and, hence, it will lie on the horizontal segment drawn through its 2-boundary, i.e., on the boundary of a cell.

Finally, property 4 follows from the fact that each slab contains at most $\sqrt{n}$ 1-boundaries and each cell at most $\sqrt{n}$ 2-boundaries. □

We will use the cells of this partition as leaves of the partition tree. For each slab s we construct a binary tree T_s that contains its cells ordered by x_2-coordinate in its leaves. Next we construct a tree T that contains the slab trees T_s ordered by x_1-coordinate in its leaves. See Fig. 2 for an example. Each node in the tree, consisting of T and the slab trees T_s, has an associated region, being the union of the cells at the leaves in the subtree. It is easy to see that such a region is a (possibly infinite) rectangle.

LEMMA 3.2. *Let V be a set of n rectangles in the plane. There exists a partition tree for storing any subset of V such that*

1. *The tree has $O(n)$ nodes.*
2. *Each rectangle is stored in $O(\sqrt{n})$ leaves.*
3. *Each rectangle influences $O(\sqrt{n}\log n)$ TOT fields.*

4. *No cell of a leaf contains vertices of rectangles in the interior.*
5. *Each leaf stores no more than $O(\sqrt{n})$ rectangles.*

Proof. Properties 1, 2, 4, and 5 follow immediately from the above lemma. The third property follows from the first two. If the tree has $O(n)$ nodes, its depth is bounded by $O(\log n)$. When a rectangle influences the TOT field of a node δ it partially covers $C_{father(\delta)}$, and there must be a leaf below $father(\delta)$ that is intersected by the rectangle. Hence, the number of internal nodes intersected by a rectangle is bounded by $O(\log n)$ times the number of leaves where the rectangle is stored. As a result, the rectangle can only influence that number of TOT fields. □

It remains to show how the measure at a leaf is maintained when inserting and deleting rectangles. Note that, due to property 4, the rectangles stored at a leaf δ are 1-piles or 2-piles with respect to C_δ. In other words, they form a trellis. The measure of such a trellis can be maintained in the following way. Let V_1 be the projection of the 1-piles on the x_1-axis and let V_2 be the projection of the 2-piles on the x_2-axis. Let M_1 be the (one-dimensional) measure of V_1 and M_2 the (one-dimensional) measure of V_2. Assume that the cell C_δ has measure $L_1 \times L_2$. Now the measure of the trellis is $L_1 \times L_2 - (L_1 - M_1) \times (L_2 - M_2)$. This follows from the fact that a point inside C_δ is covered unless it is covered in neither M_1 nor M_2. Hence, we just have to maintain the one-dimensional measure of V_1 and V_2. For this we can use a simple segment tree that uses linear storage and maintains the measure in time $O(\log n)$ per insertion and deletion (see [7]). So with each cell (leaf) δ we associate two segment trees S_1 and S_2. S_1 contains the projections of the 1-piles in δ on the x_1-axis and S_2 contains the projections of the 2-piles on the x_2-axis. Inserting (deleting) a rectangle at δ now consists of inserting (deleting) it in S_1 or S_2 (never in both). In this way, M_1 and M_2 get updated and we obtain the new measure in the leaf.

THEOREM 3.1. *The measure of a set of n 3-boxes in three-dimensional space can be computed in time $O(n\sqrt{n}\log n)$ using $O(n\sqrt{n})$ storage.*

Proof. We use the plane-sweep approach and maintain the partition tree described above. To insert or delete a rectangle, we have to update $O(\sqrt{n}\log n)$ TOT fields. This takes time $O(\sqrt{n}\log n)$. Next, we have to insert or delete the rectangle at $O(\sqrt{n})$ leaves. At each such leaf this causes an insertion or deletion in a segment tree which takes $O(\log n)$ time. Hence, the total update time of the partition tree is $O(\sqrt{n}\log n)$.

The bound on the amount of storage required follows from the fact that the tree itself takes $O(n)$ storage and each leaf stores $O(\sqrt{n})$ information (according to property 5 of the above lemma). □

In §5 we will show how to reduce the amount of storage used to $O(n)$.

4. Dynamic measure in multidimensional space. We will now generalize this method to d-dimensional space. To this end we will describe a d-dimensional partition tree, based on a cell decomposition of the d-dimensional space.

DEFINITION 4.1. A *slab at level* $i = 1, \cdots, d$ is a subset of $\mathcal{R}^d$ of the form $I_1 \times I_2 \times \cdots \times I_i \times \mathcal{R}^{d-i}$, where $I_1, \cdots, I_i$ are intervals of $\mathcal{R}$.

Let V be the set of all d-boxes that will be inserted or deleted in the partition tree. We split the x_1-axis in $2\sqrt{n}$ intervals, each of whose interior contains at most $\sqrt{n}$ 1-boundaries of boxes. This splits the d-dimensional space in $2\sqrt{n}$ slabs at level 1. For each such slab s let V_s be the set of d-boxes that partially cover s. We split V_s in two subsets: V_s^1 of d-boxes that have a 1-boundary inside s and V_s^2 of d-boxes that do not have a 1-boundary inside s. Note that $|V_s^1| \leq \sqrt{n}$. Each slab s we now split with respect to the second coordinate. We split it at the 2-boundaries of each d-box in V_s^1 and we split it at every $\sqrt{n}$th 2-boundary of d-boxes in V_s^2. As a result, we

split each slab s into $O(\sqrt{n})$ slabs at level 2. For each such slab s' let $V_{s'}$ be the set of d-boxes that partially cover s'. We again split $V_{s'}$ into two subsets: $V_{s'}^1$ of boxes that have a 1- or a 2-boundary intersecting $\mathrm{Int}(s')$ and $V_{s'}^2$ of boxes that have neither a 1- nor a 2-boundary intersecting $\mathrm{Int}(s')$. (Note that there are no boxes that have both a 1- and 2-boundary intersecting $\mathrm{Int}(s')$.) Again, $|V_{s'}^1| = O(\sqrt{n})$. We split s' into slabs at level 3 with respect to the third coordinate. Again, we split at each 3-boundary of boxes in $V_{s'}^1$ and at every $\sqrt{n}$th 3-boundary of boxes in $V_{s'}^2$. In this way we continue for all coordinates.

LEMMA 4.1. *The partition has the following properties:*

1. *There are $O(n^{d/2})$ cells.*
2. *Each d-box of V partially covers at most $O(n^{(d-1)/2})$ cells.*
3. *Each cell only contains piles in its interior.*
4. *Each cell has at most $O(\sqrt{n})$ d-boxes partially covering it.*

Proof. For each coordinate, every slab at level i is split into $O(\sqrt{n})$ slabs at level $i+1$. Hence the total number of cells we obtain is $O(\sqrt{n}^d) = O(n^{d/2})$.

If a d-box R partially covers a cell C, then an i-boundary B of R cuts through C for some i $(1 \le i \le d)$. At the moment we split slabs at level $i-1$ with respect to the ith coordinate, there are $O(n^{(i-1)/2})$ slabs. Each of these slabs is split into $O(\sqrt{n})$ slabs at level i at this moment but B can cut through only one of them (because the cutting is done with respect to the ith coordinate axis). So after the ith step, B still cuts through at most $O(n^{(i-1)/2})$ slabs at level i. In the next $(d-i)$ steps each slab at level i is cut into $O(n^{(d-i)/2})$ cells. So B will cut through at most $O(n^{(i-1)/2} \times n^{(d-i)/2}) = O(n^{(d-1)/2})$ cells.

Property 3 follows from the fact that no d-box can have both an i_1- and an i_2-boundary intersecting a slab at level i with $i_1 < i_2 \le i$. Hence, no d-box has boundaries in more than one coordinate intersecting a slab at level d, i.e., a cell. So each d-box forms a pile in a cell.

The last property follows immediately from the way we split the slabs. □

We will use the cells of this partition as leaves of the partition tree. It is easy to see how the rest of the tree can be built on top of it. The tree consists of d "stages," where each stage consists of $O(\log n)$ levels of the tree. The top stage consists of a tree T that stores the $2\sqrt{n}$ slabs at level 1 in its leaves, sorted on the x_1-coordinate. Each slab is represented by a slab tree that stores its slabs at level 2 (created in the second step) sorted by x_2-coordinate. For each of these slabs there is again a slab tree that stores its subdivision by x_3-coordinate, etc.

LEMMA 4.2. *Let V be a set of n d-boxes in d-dimensional space. There exists a partition tree for storing any subset of V such that*

1. *The tree has $O(n^{d/2})$ nodes.*
2. *Each d-box is stored in $O(n^{(d-1)/2})$ leaves.*
3. *Each d-box influences $O(n^{(d-1)/2} \log n)$ TOT fields.*
4. *Each cell of a leaf only contains piles.*
5. *Each leaf stores no more than $O(\sqrt{n})$ d-boxes.*

Proof. Properties 1, 2, 4, and 5 follow immediately from the above lemma. The third property follows from the first two as the depth is again bounded by $O(\log n)$. □

It remains to show how the measure at a leaf is maintained when inserting and deleting d-boxes. As stated in property 4, the d-boxes stored at a leaf δ are piles and form a d-dimensional trellis. Let V_i be the projection of the i-piles on the x_i-axis for each $1 \le i \le d$. Let M_i be the one-dimensional measure of V_i. Let L_i be the length

of C_δ in direction x_i. The following result is easy to prove.

LEMMA 4.3. *The measure of the trellis is*

$$\prod_{1\le i\le d} L_i - \prod_{1\le i\le d} (L_i - M_i).$$

When M_i is known for each i, the measure can be computed in constant time (assuming d is a constant).

Hence, we just have to maintain the one-dimensional measure of V_i for each i. For this we use d segment trees $S_1 \cdots S_d$, one for each dimension. An insertion or deletion in a leaf means inserting or deleting the i-pile in the correct segment tree S_i. In this way we obtain the updated measure M_i and we can recompute the above formula to obtain the new measure in the cell. This will take time $O(\log n)$ (assuming that d is a constant).

LEMMA 4.4. *Updates in the d-dimensional partition tree take time* $O(n^{(d-1)/2} \log n)$ *and the tree uses* $O(n^{(d+1)/2})$ *storage.*

Proof. The proof follows from the above lemmas. □

THEOREM 4.1. *The measure of a set of n d-boxes in d-dimensional space can be computed in time* $O(n^{d/2} \log n)$ *using* $O(n^{d/2})$ *storage.*

Proof. We use the plane-sweep approach and maintain a $(d-1)$-dimensional partition tree. So we have to perform $O(n)$ updates, each taking time $O(n^{(d-1-1)/2} \log n)$. The time bound follows. According to the preceding lemma, the structure uses $O(n^{(d-1+1)/2})$ storage. □

5. Reducing the amount of storage. In this section we will show how the amount of storage required can be reduced to $O(n)$. To this end we use an instance of the streaming technique introduced in Edelsbrunner and Overmars [3].

The idea of streaming is the following: Beforehand we know what updates have to be performed and in what order. We can view the space-sweep method as traversing in time (being the dth coordinate). Each update in the structure has to be performed at a specific moment in time. Before each update we check what the current measure is and we multiply it by the time passed since the last update. Rather than building the structure and performing the updates one after the other, we will perform them simultaneously and construct parts of the data structure when we need them. When we are ready with the part we discard it again to free memory.

To formalize this, at any moment we are given a sequence of updates L over time and a region of the space C. This region corresponds to some node in the tree and L is the sequence of updates that will pass through this node. With each update in L we have stored the time at which it has to be performed. In the beginning C is the whole $(d-1)$-dimensional space and L is the complete list of updates, time being the dth coordinate. A counter $MEAS$ will be used to collect all the measure found. In the beginning it will be set to 0.

The technique now works as follows: When all $(d-1)$-boxes in L are piles with respect to C (i.e., we are at a leaf in the partition tree) we construct $d-1$ segment trees. We perform all the updates on the segment trees and compute the $(d-1)$-measure in the cell after each update. These measures we multiply with the time period to the next update to obtain the d-measure in C. This d-measure we add to $MEAS$. This will take time $O(|L| \log n)$ and storage $O(|L|)$. Afterwards we destroy all the structures.

When not all boxes are piles (i.e., we are at an internal node) we first compute during which periods of time C will be completely covered by one box. (This corresponds to the time when $TOT \neq 0$.) This can be done by simply walking along the list of updates and maintaining the number of boxes that cover C. Whenever this number is larger than 0, C is covered. This takes time $O(|L|)$. We multiply the $(d-1)$-measure of C with the total amount of time C is covered and add it to $MEAS$. Next we change time by collapsing the covered periods into a single moment, performing all the updates in that period at the same moment. (This is necessary to avoid measure being found lower in the tree during these periods again and counted twice.) Boxes that are now inserted and deleted at the same moment are removed from L. Again, this takes time $O(|L|)$ only.

Next we split C into two cells, C_1 and C_2, in a way similar to how it would have been split in the partition tree. This can be done in the following way. Remember that in the first stage of the tree we split on the x_1-coordinate, in the next stage on the x_2-coordinate, etc., until, in the last stage we split on the x_{d-1}-coordinate. Hence, it is easy to remember on which coordinate we have to split at a particular moment. So assume we have to split on the ith coordinate. We make two different splits: splits along i-boundaries in V_C^1 or splits along i-boundaries of boxes in V_C^2 (see the previous section). There is no problem in first making the splits along i-boundaries of V_C^1 and after that along i-boundaries in V_C^2. (The tree will get a depth that is at most twice as large.) So making a split can be done as follows:

- Let i be the current splitting coordinate. Split L into V_C^1 and V_C^2.
- If $V_C^1 \neq \emptyset$, then split along the median i-boundary in V_C^1.
- Else, if V_C^2 contains more than $\sqrt{n}$ i-boundaries split along the median i-boundary in V_C^2.
- Else, increase i and repeat the procedure.

Finding the splitting line can easily be done in time $O(|L|)$. It is easy to see that the resulting partition tree will still satisfy the properties in Lemma 4.2.

Now we construct the list L_1 out of L containing the updates that influence C_1. In L we only keep the other updates. Hence, each update is either stored in L_1 or in L. We recursively call the routine for C_1 and L_1. When we get back from the recursive calls, we join L_1 and L to reconstruct L in its original form. Now we determine the list L_2 of updates that influence C_2, again leaving in L the other updates. We now recursively call the routine with C_2 and L_2. When we get back we again reconstruct L (to be used one level higher in the recursion). Note that during the whole process we never copy updates. We simply take a part of list L and send it down the recursion. When we get back we take another part of L and go again in recursion. As a result, each update is stored at at most one place.

The method does essentially the same work as the original technique, in which all updates are performed one after the other. In fact, it is more efficient for two reasons. When the whole list consists of piles, we immediately solve the problem rather than splitting till the list contains fewer than $\sqrt{n}$ boxes. Second, we do not consider boxes anymore when, during their whole period of existence, they are covered by some other box.

THEOREM 5.1. *The measure of a set of n d-boxes in d-dimensional space can be computed in time $O(n^{d/2}\log n)$ using $O(n)$ storage.*

Proof. The amount of time used is essentially the same as when we performed the updates one after the other.

To estimate the amount of storage, note that each update is stored at most once

in a list L. The bound follows. □

6. Extensions. The partition tree and method described above can also be used to solve a number of related problems. In this section we will briefly mention some of them.

It is well known that the perimeter of the union of n rectangles in the plane can be computed in time $O(n \log n)$. (See, e.g., [7], [9].) Computing the perimeter generalizes to computing the $(d-1)$-dimensional measure of the contour of the union of a set of d-boxes in d-dimensional space. The contour consists of parts of i-boundaries of boxes that do not lie in the interior of the union. We will only show how to compute the measure of the parts of d-boundaries of the contour. The measure of the i-boundaries for other i can easily be obtained by renumbering coordinates. The total measure of the boundary is obviously the sum of the measures of the i-boundaries in the contour for all $1 \le i \le d$.

To compute the measure of the d-boundaries of the contour we use exactly the same method as in §4. We move a sweep plane along the dth coordinate axis and maintain the measure of the intersection. At any d-boundary where the sweep-plane halts, we update the $(d-1)$-dimensional measure as in §4. The part of this d-boundary that is part of the contour is obviously the absolute value of the difference between the old and the new measure, except when more boundaries have the same dth coordinate value. (In this case, some care has to be taken. The procedure below correctly treats those cases.) To be precise, the main algorithm (as described in §2) is changed as follows:

S:=$\emptyset$;
$MEAS$:=0;
for i:=1 **to** n' **do**
  M:=$(d-1)$-dimensional measure of boxes in S;
  Insert all d-boundaries of boxes that start at a_i in S;
  M_+:=$(d-1)$-dimensional measure of boxes in S;
  Delete all d-boundaries of boxes that end at a_i from S;
  M_-:=$(d-1)$-dimensional measure of boxes in S;
  $MEAS$:=$MEAS + (M_+ - M) + (M_+ - M_-)$
end;

S is again stored as a partition tree and maintained in exactly the same way. The correctness of the method is easily established. This leads to the following result:

THEOREM 6.1. *The measure of the contour of the union of a set of n d-boxes in d-dimensional space can be computed in time $O(n^{d/2} \log n)$ using $O(n^{d/2})$ storage.*

The method can also be used to compute the measure of lower-dimensional parts of the contour. It is unclear how streaming can be applied here to reduce storage.

As a second application, consider the following query problem: Given a set of d-boxes in d-dimensional space, store them such that for a given query box R it can efficiently be determined whether R is completely covered by the d-boxes.

To solve this problem we store the d-boxes in a d-dimensional partition tree. A query is performed using the procedure "filled," described below. It gets two arguments, a node δ and the query rectangle R, and returns whether the part of R inside C_δ is fully covered by d-boxes. Calling the routine with δ the root of the tree gives the required answer.

procedure filled (δ, R):**boolean**;
  if R completely covers C_δ **then**

```
        return M_δ = measure of C_δ
    else if R partially covers C_δ then
        if δ is a leaf then
            search the segment trees to see whether in at least
            one of them the projection of R is fully covered;
            return the result
        else
            return filled(lson_δ,R) and filled(rson_δ,R)
    else
        return true
end;
```

The correctness of the method can easily be established. Searching the segment trees in a leaf takes time $O(\log n)$. This has to be done in at most $O(n^{(d-1)/2})$ leaves. The total number of internal nodes visited is bounded by $O(n^{(d-1)/2} \log n)$. Theorem 6.2 follows.

THEOREM 6.2. *Let V be a set of n d-boxes in d-dimensional space. One can store V using $O(n^{(d+1)/2})$ storage, such that for a given d-box R one can determine in time $O(n^{(d-1)/2} \log n)$ whether R is completely covered by the boxes in V.*

The method can easily be extended to compute the measure of the union of the d-boxes restricted to a given box R, in the same time bounds. Updates from a fixed set of boxes can be performed in time $O(n^{(d-1)/2} \log n)$ using the same method as described for maintaining the measure.

Other applications exist. For example, it is possible to use the techniques given here to determine contours and i-contours (contour of the area covered by at least or precisely i d-boxes).

7. Conclusions. We have given a new solution to Klee's measure problem that is much more efficient than previously known results, improving the time bound from $O(n^{d-1})$ to $O(n^{d/2} \log n)$. The technique uses some new ideas, including a result on partitioning space, a new type of partition tree, and the use of trellises. Streaming was applied to reduce the amount of storage used to $O(n)$.

The dynamic data structure we presented for dynamically maintaining the measure can be used for other problems as well. As we have shown, it is very simple to compute, e.g., the perimeter. Moreover, the structure gives a compact representation of the shape of the set of d-boxes. This can be used to answer certain classes of queries efficiently.

Some open problems remain. First, it might be possible to shave off the factor of $\log n$. But, in fact, there is no reason to believe that the method is even near optimal. Improvements or lower bounds should be worked on. It is also interesting to look at the measure of other objects. For example, the best bound known for computing the measure of the union of a set of triangles is $O(n^2)$.

REFERENCES

[1] J. L. BENTLEY, *Multidimensional binary search trees used for associated searching*, Comm. ACM, 18 (1975), pp. 509–517.

[2] ———, *Algorithms for Klee's rectangle problem*, unpublished notes, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1977.

[3] H. EDELSBRUNNER AND M. H. OVERMARS, *Batched dynamic solutions to decomposable searching problems*, J. Algorithms, 6 (1985), pp. 515–542.

[4] M. L. FREDMAN AND B. WEIDE, *The complexity of computing the measure of* $\cup[a_i, b_i]$, Comm. ACM, 21 (1978), pp. 540–544.

[5] V. KLEE, *Can the measure of* $\cup[a_i, b_i]$ *be computed in less than* $O(n \log n)$ *steps?*, Amer. Math. Monthly, 84 (1977), pp. 284–285.

[6] M. H. OVERMARS, AND C.-K. YAP, *New upper bounds in Klee's measure problem (extended abstract)*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, White Plains, NY, 1988, pp. 550–556.

[7] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, Berlin, New York, 1985.

[8] J. VAN LEEUWEN AND D. WOOD, *The measure problem for rectangular ranges in d-space*, J. Algorithms, 2 (1980), pp. 282–300.

[9] P. M. B. VITÁNYI AND D. WOOD, *Computing the perimeter of a set of rectangles*, Tech. Report 79-CS-23, Unit for Computer Science, McMaster University, Hamilton, Ontario, Canada, 1979.

[10] E. WELZL, *Partition trees for triangle counting and other range searching problems*, in Proc. 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1988, pp. 23–33.

[11] D. E. WILLARD, *Polygon retrieval*, SIAM J. Comput., 11 (1982), pp. 149–165.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1046–1067, December 1991

© 1991 Society for Industrial and Applied Mathematics
005

AN OPTIMAL RANDOMIZED PARALLEL ALGORITHM FOR FINDING CONNECTED COMPONENTS IN A GRAPH*

HILLEL GAZIT†

Abstract. A parallel randomized algorithm for finding the connected components of an undirected graph is presented. The algorithm has an expected running time of $T = O(\log(n))$ with $P = O((m+n)/\log(n))$ processors, where m is the number of edges and n is the number of vertices. The algorithm is *optimal* in the sense that the product $P \cdot T$ is a linear function of the input size. The algorithm requires $O(m+n)$ space, which is the input size, so it is *optimal* in space as well.

Key words. connected components, randomized algorithm, parallel computing, CRCW, undirected graph

AMS(MOS) subject classifications. 05C38, 05C40, 68Q25, 68Q10, 68R10

1. Introduction. In this paper, the problem of finding the connected components in an undirected graph $G = (V, E)$ is considered. There are several well known and fast sequential algorithms like Depth First Search (DFS) and Breadth First Search (BFS) for finding connected components. This paper presents a solution to the problem on a parallel model: the Concurrent-Read Concurrent-Write (CRCW) Parallel Random Access Machine (PRAM). It is a synchronized parallel-computation model where all the processors can read and write into a common memory. In the case of concurrent writes into the same memory location, it is assumed that an arbitrary processor succeeds. The algorithm has an expected running time of $O(\log(n))$, using $O((m+n)/\log(n))$ processors, where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges. The probability that the algorithm runs longer than expected is at most $(2/e)^{n/\log^k(n)}$ for some constant $k < 4$. The algorithm is as fast as the best-known algorithms [11], [10], [3] and is optimal in the sense that $P \cdot T$ is equal to the complexity of the sequential algorithm.

Shiloach and Vishkin [11] conjecture that the barrier of $\log(n)$ cannot be surpassed by any polynomial number of processors. Assuming their conjecture holds, we have achieved the lower bound for running time with an optimal number of processors. The question of whether these bounds can be achieved by a deterministic algorithm is still open.

The algorithm presented here can also be used to find a spanning forest, which is needed for most algorithms for two-connectivity, three-connectivity, and four-connectivity [12], [9], [8].

We approach the problem of finding the connected components in an undirected graph by solving three successively harder cases: *easy graph*, *dense graph*, and *sparse graph*. A solution for the *easy graph* case (so called because it allows a processor for every vertex and a processor for every edge) was given by Shiloach and Vishkin [11]. A variant of their algorithm is presented in §3, Fig. 1. The *dense graph* case is so called because it allows a processor for every vertex but only one processor per $\log(n)$ edges. This problem can be reduced in $O(\log(n))$ time to the *easy graph* case. The reduction is given in §4, Fig. 2. The *sparse graph* case is the most difficult one. It allows only one processor per $\log(n)$ vertices and one processor per $\log(n)$ edges. This problem can be reduced in $O(\log(n))$ time to the *dense graph* problem, by repeatedly finding

* Received by the editors May 31, 1988; accepted for publication (in revised form) January 4, 1991. This work was supported by National Science Foundation grant DCR-8514961.

† Department of Computer Science, Duke University, Durham, North Carolina 27706.

partial connected components of the graph until the number of connected components is equal to the number of processors. The reduction is given in §5, Fig. 5. Thus we can solve the general problem in $O(\log(n))$ time and optimal number of processors.

The last two algorithms described are randomized. In a randomized algorithm, each processor has access to a random-number generator which returns random numbers of $\log(\log(n))$ bits in unit time. The main idea in the reduction algorithms is to find a way to separate vertices with a large number of incident edges, called *extrovert* vertices, from those with a small number of incident edges, called *introvert*. To separate them by counting the edges takes too much time. Therefore a statistical test is used: a sample of edges is taken and only the vertices they hit are considered. Obviously, an *extrovert* vertex is more likely to be chosen by this method because it has more edges than an *introvert* vertex.

2. Preliminaries.

2.1. Previous results.

The problem of finding connected components in a graph has attracted much attention in the last ten years. In 1979, Hirschberg, Chandra, and Sarwate [2] presented an $O(\log^2(n))$ parallel connectivity algorithm for a graph with n vertices and m edges. Their algorithm uses $n^2/\log(n)$ processors on the CREW model.[1] In the same year Wyllie [13] presented a more detailed version of their algorithm. Shiloach and Vishkin [11] improved the result to $O(\log(n))$ time and $O(m+n)$ processors on the CRCW model in 1982. In 1984, Reif [10] found a simple probabilistic algorithm with the same complexity as the Shiloach–Vishkin algorithm. In 1986, Cole and Vishkin [3] presented a deterministic CRCW algorithm of $O(\log(n))$ time using $O(((m+n)/\log(n))\cdot\alpha(m,n))$ processors, where $\alpha(m,n)$ is an inverse Ackerman function.

All these algorithms share the same basic problem, which also explains why the number of processors is not optimal. In the early iterations the number of vertices is reduced in each iteration, but the order of the number of edges may remain the same. Therefore it is hard to accelerate the algorithm in the first iterations. One exception can be found in the case of a planar graph, where the number of edges is bounded by three times the number of vertices. Using this fact, Hagerup [7] gave an optimal deterministic connectivity algorithm for a planar graph. However, the space complexity of his algorithm is not linear.

In 1986, Gazit [6] presented an optimal randomized connectivity algorithm. This paper contains a new version of that algorithm, with improved probability; in the new version the probability of failure is exponentially inverse to the input size.

2.2. Definitions.

An undirected graph $G=(V,E)$ consists of a set of *vertices* V of size n and a set of *edges* E of size m. Each edge is an unordered pair (v,w) of disjoint vertices v and w. In this paper it is assumed that there may be more than one edge between two vertices. The edge (v,w) *hits* vertices v and w. The *degree* of a vertex v is the number of edges that hit v. A *path* joining v_1 and v_k in G is a sequence of vertices $v_1, v_2, \cdots, v_k$ such that $(v_i, v_{i+1})\in E$. A *partial connected component* in a graph is a subset $U\subseteq V$, such that for every pair of vertices v, $u\in U$, there is a path in U joining v and u. A *connected component* is a partial connected component that is not contained in any other partial connected component. A vertex u is *isolated* with respect to some vertex set V if there is no edge from u to any vertex $v\in V$.

The following notations are used:

[1] A preliminary version of their algorithm was presented in 1976 at the ACM Symposium on Theory of Computing, Hershey, PA.

1. For convenience, define $\alpha = \sqrt{\frac{2}{3}}$.
2. $P =$ Number of processors.
3. $n =$ Number of vertices.
4. $m =$ Number of edges.

Assuming there are at least $(n+m)/\log(n)$ processors, the following definitions are given:

- An *easy graph* is a graph with $n+m \leq P$.
- A *dense graph* is a graph with $n \leq P$.
- A *sparse graph* is a graph with $n > P$.

2.3. Data structure. The main data structure is a reverse-rooted tree called *supervertex*. Every vertex has a unique pointer to its parent. Hence there is a directed path from every vertex in the tree to the root of that tree. The *root* for vertex v, denoted *root*(v), is defined to be the root of the supervertex tree to which v belongs. There are no cycles in a supervertex tree, but each includes one self-loop: the root of each tree points to itself.

We will see later that this data structure is a natural one for solving the connected components problem and, indeed, it was used in previous algorithms [11], [3].

A supervertex of height 0 or 1 is a *star*.

Supervertices are viewed as vertex-disjoint partial connected components. At any stage of the algorithm, the graph is a forest of reversed-rooted trees. An edge (u, v) is *live* if u and v belong to different supervertices. (There can be more than one edge between two supervertices.) A supervertex is *live* if there is a live edge incident on one of its vertices. The *degree* of a supervertex is the number of live edges that hit its vertices.

These supervertices are a reverse-rooted forest of vertices. Forming the union of two supervertices makes the root of one supervertex a child of a vertex of the other supervertex. Each vertex v has a pointer, *parent*(v), to its parent in the supervertex tree.

Every vertex v has a single bit flag called ext_v, a memory cell e_v to store an edge, and a memory cell $flag_v$ that can hold an integer of size $2 \cdot \log(\log(n))$.

3. The easy case. In this section, we describe an algorithm for finding connected components when there is a processor for every edge and a processor for every vertex. This is a variant of Shiloach and Vishkin's algorithm [11]; an outline is presented in Fig. 1.

The algorithm develops an inverse rooted tree for each partial connected component. When all such trees are stars and there are no more live edges, the components are maximal.

Each iteration performs two operations:

1. Reducing tree height.
2. Joining trees.

Reducing tree height is a simple matter of "jumping over" a vertex; each parent pointer is reset to point to the grandparent. A tree that had height h before the jump-over has height $\lceil \frac{h}{2} \rceil$ after it. Since the root is its own parent, stars are not affected; other trees are reduced in height by a factor between $\frac{1}{3}$ and $\frac{1}{2}$. Height reduction is Step 1 and Step 4 of each iteration.

Joining trees is more complicated. There are two cases. The first (Step 2) pertains when the live edge connects vertices that are children of their respective roots. In this case, to avoid creating cycles (i.e., to preserve the data structure), we make the

```
procedure easy-case (V, E)
for all v ∈ V in parallel do parent(v) := v od
while there is a live edge in the graph or some tree is not a star do
/* Step 1 */
    for all v ∈ V in parallel do
        parent(v) := parent(parent(v))
    od
    for every live edge (u, v) using concurrent write in parallel do
/* Step 2 */
        if parent(parent(v)) = parent(v) and parent(parent(u)) = parent(u)
        then if parent(u) > parent(v) then parent(parent(u)) := parent(v)
                else parent(parent(v)) := parent(u)
             fi
        fi
/* Step 3 */
        if parent(u) = parent(parent(u)) and
          parent(u) did not get new incoming edges in Steps one and two
        then parent(parent(u)) := parent(v)
        fi
        if parent(v) = parent(parent(v)) and
          parent(v) did not get new incoming edges in the first two steps
        then parent(parent(v)) := parent(u)
        fi
    od
/* Step 4 */
    for all v ∈ V in parallel do
        parent(v) := parent(parent(v))
    od
od

end easy-case
```

FIG. 1. *Finding connected components: the easy case.*

higher-numbered root the child of the lower. The second case (Step 3) involves a star not affected previously in the current iteration and having a live edge. The root of such a star becomes a sibling of the other vertex hit by that edge.

We will prove later that the total height of the live trees is reduced by a constant factor in every iteration. Therefore, after $O(\log(n))$ iterations the height is at most 1 and the algorithm stops because no live stars, or trees which are not stars, remain. Some results from [11], which will be needed later in this paper, are presented here. The detailed proofs can be found in the original paper.

LEMMA 3.1. *Before using an edge in Step* 2 *or* 3, *determining whether it is live takes* $O(1)$ *time using one processor.*

Proof. Consider some edge (u, v). If we are to use it in Step 2, both u and v are children of their respective roots, so we can determine immediately whether it is live. If we are to use it in Step 3, one vertex, call it v, must be the child of a root that got no new edges in this iteration. If u is also the child of its root, determination is immediate. If not, then u is not in a star; therefore $root(u)$ got a new edge in Step 1, so $root(u)$ cannot also be $root(v)$, which is to say that the edge is live. □

LEMMA 3.2. *During the execution of the algorithm, the partial connected components are always kept as reverse rooted trees.*

Proof. We prove this by induction. The lemma is true before the **while** loop starts because every vertex is itself a reverse-rooted tree of one vertex, and is itself a

partial connected component.

In every iteration we link only trees that have some edges between them and therefore belong to the same partial connected component.

It is also clear that the data structure remains a forest. The height reduction process changes a grandparent into a parent, creating no new cycles. Nor can the join operations create cycles. A Step 2 join links roots only, making the higher-numbered the child of the lower. This consistency prevents cycles. In a Step 3 join, one supervertex must be a star, so all of its edges point to the root. Resetting the root's parent pointer to a vertex in a different tree cannot create a cycle. □

LEMMA 3.3. *If a tree T has not been changed during an entire iteration, it remains unchanged until the end. This tree is a star and its vertices are a maximal connected component.*

Proof. Every unchanged tree must be a star, otherwise it would have some vertex v of depth 2, and then the operation $parent(v) := parent(parent(v))$ would change the tree. By Lemma 3.2 this star represents a partial connected component. It is maximal; otherwise it would have a live edge e that would be used in the second or third step, thus changing it. □

DEFINITION 3.4. The height of the forest is the *sum* of the heights of all the trees that are either live or have height greater than one.

LEMMA 3.5. *After the first iteration, the height of the forest is at most $(2 \cdot n)/3$.*

Proof. After the first three steps, every vertex with a live edge is in a tree with height one or more. The total height of all the stars is bounded above by half the number of vertices in the stars (we may reach the upper bound if every star has only two vertices). The total height of the nonstar trees is bounded by the number of vertices in these trees. After the fourth step, each of those trees has at most $\frac{2}{3}$ of its original height. □

LEMMA 3.6. *The height of the forest is reduced by a factor of at least $\frac{1}{3}$ in each subsequent iteration.*

Proof. If a tree is not a star, then the first step reduces its height to at most $\frac{2}{3}$ of its original height. The second and third steps do not increase the height of the forest, because we link roots to internal vertices (not leaves) of other trees.

The first and fourth steps do not change the height of the stars, but every live star is linked by either the second step or the third step. If a tree of height h and a star are linked together, in the worst case the height of the new tree is $h+1$. After the fourth step, this height is at most $\frac{2}{3} \cdot (h+1)$. □

COROLLARY 3.7. *The number of the live trees after iteration i is bounded by $n \cdot (\frac{2}{3})^i$.*

Proof. The corollary follows immediately from Lemma 3.6, because the minimum height of each live tree is one. □

THEOREM 3.8 (Main Theorem of [11]). 1. *The algorithm terminates after $\lceil \log_{\frac{3}{2}}(n) \rceil$ iterations.*

2. *$parent(u) = parent(v)$ if and only if u and v are in the same connected component.*

Proof. By Lemmas 3.5 and 3.6 the height of the forest after $\lceil \log_{3/2}(n) \rceil$ iterations is at most one. That means that there is at most one live supervertex and that supervertex is a star. Therefore, there cannot be any live edges, so obviously there are no live supervertices.

Subsequently, each tree is unchanged by further iterations because it is a dead star. Therefore, by Lemma 3.3, each tree represents a maximal connected component. □

4. Dense-to-easy reduction.

4.1. Informal description. We define a dense graph as a graph for which there is a processor for every vertex and a processor for every $\log(n)$ edges.

We no longer have enough processors to check every edge in the graph in every iteration, but, as we have a processor for every vertex, we can still reduce the height of the supervertices. Therefore, our two basic operations in every iteration will be:

1. Reducing the height of all the trees.
2. Joining together some trees with a large number of live edges.

We prefer to join supervertices with many live edges because once all of these supervertices are joined, all the live edges remaining in the graph will have at least one endpoint in a supervertex of low degree; therefore the number of live edges will be of the same order as the number of vertices.

We need a means of deciding which supervertices have a high degree, with respect to live edges. We also need some method of finding at least one live edge for most stars with high degree.

The method presented here repeatedly chooses a random sample of edges. A supervertex of high degree (a dense supervertex) has a higher probability of having at least one of its incident edges chosen than a supervertex of low degree (a sparse supervertex). The goal is to join dense supervertices among themselves and leave out the sparse ones.

It is possible that a supervertex has a live edge that was not chosen in some iteration; but as the number of its live edges increases, so does the probability that at least one of them will be chosen. Therefore, the probability that a given join involves a dense supervertex, is greater than the probability that it involves a sparse one.

After the *dense-to-easy* algorithm is completed, all the remaining live edges are connected to sparse supervertices. It will be shown later that the expected number of live edges in the graph after the algorithm has been executed is $O(n)$, so that the *easy-case* algorithm can be used.

Every iteration is composed of four steps, similar to the steps of the *easy-case* algorithm. In the first and the last iteration we reduce the height of all the trees. These operations can be done in constant time using $O(n)$ processors.

In the join steps we use a *subset* of edges. If a star has at least one edge to a dense vertex in our subset, we can use that edge to link it to another tree. If the star has an edge to a dense vertex with a smaller identification number, we link it in Step 2. If the star was not changed in the first two steps and it has an live edge to a dense vertex, we link it in Step 3.

If a star was not changed in the entire iteration, then either it is a dead star or none if its edges was sampled. We assume that the degree of that star is low and set it aside, even if one of its edges is sampled later.

As noted above, when selecting a supervertex by edge sampling, the probability that a sampled edge hits this supervertex is higher if the supervertex has many edges. If one randomly picks a large enough sample of edges and considers some supervertex v, and none of these edges hits the supervertex, then there is a high probability that v is sparse.For each edge in the sample, an attempt is made to join its end-vertices. In the next iteration another (smaller) sample is chosen, and the process is repeated. Although several edges in the sample may hit the same supervertex, only one of them may be used to join the root to another tree.

To reduce the time complexity of successive iterations, the size of the sample we choose decreases geometrically. This means that the probability that a given supervertex has no incident edge (and therefore is classified as sparse) increases. We will show later that the expected number of live edges hitting a supervertex classified as sparse is inversely proportional to the sample size.

The upper bound for the number of dense supervertices also decreases geometrically. We choose the samples so that the sample size decreases more slowly than the number of supervertices, achieving two desirable results:

- For each iteration, consider the set of all supervertices that were classified as sparse during this iteration and the expected total number of live edges that hit this set; we will show that this number decreases geometrically.
- The time complexity of each step is proportional to the sample size. The sample size decreases geometrically and the time complexity is equal to

$$\frac{m}{P} = O(\log(n));$$

therefore the time complexity is bounded by the *easy-case* algorithm time complexity.

The algorithm presented in Fig. 2 reduces a dense graph to the easy graph case. After execution of the algorithm, every vertex in the graph points to the root of the supervertex that represents the partial connected component to which it belongs. All edges connect roots; and the expected number of live edges is $O(n)$. Therefore, the easy-case algorithm can be applied. The reduction algorithm always halts, but the number of live edges that remain in the graph at the end may be greater than expected. The time complexity is $O(\log(n))$. The probability of failure is less than $\lceil \log_{3/2}(n) \rceil \cdot (2/e)^{n/\log^3(n)}$

4.2. Analysis of the dense-to-easy reduction.

DEFINITIONS.

1. An *extrovert root* r is a root with $\text{ext}_r = 1$. At the beginning of the *dense-to-easy* procedure, every vertex is an *extrovert* root.
2. An *extrovert supervertex* is a supervertex such that its root is an *extrovert* root.
3. An *extrovert vertex* is a vertex that belongs to a tree whose root is an *extrovert* root.
4. An *introvert vertex* v is a vertex that belongs to a star and $\text{ext}_{parent(v)} = 0$. At the end of the *dense-to-easy* procedure, all the vertices are *introvert.*
5. An *extrovert edge* is a live edge that connects two *extrovert* vertices.
6. An *extrovert edge* becomes an *introvert edge* if one of its end supervertices becomes *introvert.*
7. The random variable n_i represents the number of *extrovert* supervertices after iteration i.

LEMMA 4.1. *Determining if a tree was changed in some iteration i can be done in $O(1 + ((m \cdot \alpha^i)/P))$ time.*

Proof. If a tree is not a star, then it is changed in the first step and the root gets new incoming edges. If a tree was not changed in the first step, then it was a star before the iteration started. In that case all the new incoming edges are connected directly to the root.

Using Concurrent Write, we can notify all the roots that were changed when we set the new incoming edges. ∎

```
procedure dense-to-easy (G(V, E))
for all v ∈ V in parallel do ext_v := 1 od
for all v ∈ V in parallel do parent(v) := v od
i := 0
while there is a root r such that ext_r = 1 do
/* Step 1 */
    for all v ∈ V in parallel do parent(v) := parent(parent(v)) od
    Pick a sample of edges of size max (P, ⌈m · α^i⌉)
/* Step 2 */
    for every live edge (u, v) in the sample using concurrent write in parallel do
        if ext_parent(u) = ext_parent(v) = 1 then
            if parent(parent(v)) = parent(v) and
               parent(parent(u)) = parent(u)
            then if parent(u) > parent(v) then parent(parent(u)) := parent(v)
                   else parent(parent(v)) := parent(u)
                 fi
            fi
/* Step 3 */
            if parent(u) = parent(parent(u)) and
               parent(u) did not get new incoming edges in the previous two operations
            then parent(parent(u)) := parent(v)
            fi
            if parent(v) = parent(parent(v)) and
               parent(v) did not get new incoming edges in the previous two operations
            then parent(parent(v)) := parent(u)
            fi
        fi
    od
/* Step 4 */
    for all v ∈ V in parallel do parent(v) := parent(parent(v)) od
    for every root r in parallel do
        if the tree rooted at r was not changed during the iteration
        then ext_r := 0 fi
    od
    i := i + 1
od
for every edge (u, v) in the graph in parallel do
    if (u, v) is a live edge then replace it by an edge (root(u), root(v)) fi
od

end dense-to-easy
```

FIG. 2. *Reducing a dense graph to an easy graph.*

Let S be a subset of the *extrovert* stars. Define $deg(S)$ as the number of *extrovert* edges with at least one endpoint in a supervertex in S; that is,

$$\deg(S) = |\{(u, v)|(u, v) \text{ is an extrovert edge } \wedge \ (u \in S \vee v \in S)\}$$

LEMMA 4.2. *Given a subset of stars S, such that $deg(S) = x$, the probability that none of these x edges is chosen in iteration i (i.e., that all the vertices of S and all the x extrovert edges will become introvert) is less than or equal to $e^{-x \cdot (y/m)}$, where $y = \max(P, \lceil m \cdot \alpha^i \rceil)$ is the size of the sample of edges chosen in procedure dense-to-easy.*

Proof. The probability that a particular edge is chosen by one processor is $1/m$. The probability that an edge is not chosen for the sample is bounded by $(1 - (1/m))^y$,

where y is the sample size. The probability that none of the x edges is chosen is at most:

$$\left[\left(1-\frac{1}{m}\right)^{y}\right]^{x} = \left(1-\frac{1}{m}\right)^{x\cdot m\cdot(y/m)} \leq \left(e^{-1}\right)^{x\cdot(y/m)} = e^{-x\cdot(y/m)}. \qquad \square$$

LEMMA 4.3. *For a given x, the probability that at least x edges will become introvert in iteration i is bounded by $e^{-x\cdot(y/m)}\cdot 2^{n_i}$, where n_i is the number of extrovert supervertices after iteration i and $y = \max(P, \lceil m\cdot\alpha^i\rceil)$.*

Proof. In every iteration a (possibly empty) subset of vertices becomes *introvert*. By Lemma 4.2 the probability that a particular subset of vertices with x *extrovert* edges will become *introvert* is at most $e^{-x\cdot(y/m)}$. We have at most 2^{n_i} subsets with x *extrovert* edges because a set with n_i elements has at most 2^{n_i} subsets.

We find an upper bound on the probability that at least one such subset will become *introvert* by summing up the probabilities of all these events. Therefore, an upper limit to the probability is $e^{-x\cdot(y/m)}\cdot 2^{n_i}$. $\square$

Please note that the bound is not tight:

1. We consider all subsets, including those (like the empty set) which have fewer than x edges.
2. We assume that a subset with more than x edges has the same probability of becoming *introvert* as a subset with x edges. The actual probability is smaller.
3. We sum the probabilities of events that are not mutually exclusive.

LEMMA 4.4. *The number of extrovert supervertices after iteration i is at most $n\cdot(\frac{2}{3})^i$.*

Proof. The proof is similar to the proof of Corollary 3.7. $\square$

LEMMA 4.5. *The* **while** *loop of the dense-to-easy algorithm will have terminated after at most $\lceil\log_{3/2}(n)\rceil$ iterations.*

Proof. The proof is similar to the proof of Theorem 3.8. $\square$

LEMMA 4.6. *After the* **while** *loop of the dense-to-easy algorithm has finished, all the trees are stars.*

Proof. The **while** loop does not terminate until all trees have became *introvert*, and only supervertices that are stars can become *introvert*. $\square$

DEFINITION 4.7. Let f_i be the probability that more than $n\cdot\alpha^i$ *extrovert* edges will become *introvert* during iteration i.

LEMMA 4.8. *The probability f_i is bounded by $(2/e)^{n\cdot\alpha^{2i}}$.*

Proof. Set $x = n\cdot\alpha^i$, $z = m\cdot\alpha^i$; by Lemma 4.4, $n_i \leq n\cdot\alpha^{2\cdot i}$. Note that $y \geq z$. Then, by Lemma 4.3, $f_i \leq e^{-n\cdot\alpha^i\cdot\alpha^i}\cdot 2^{n\cdot\alpha^{2\cdot i}} = (2/e)^{n\cdot\alpha^{2\cdot i}}$. $\square$

DEFINITION 4.9.

$$\gamma_n = \left\lceil -3\cdot\frac{\log(\log(n))}{\log(\alpha)}\right\rceil.$$

Note that $\alpha^{\gamma_n} \leq \log^{-3}(n)$.

LEMMA 4.10. *The probability that more than $n/\log^2(n)$ extrovert edges will become* introvert *during iteration $i \geq \gamma_n$ is bounded by $(2/e)^{n/\log^3(n)}$.*

Proof. Set $x = (n/\log^2(n))$, $y = P$, and $n_i \leq n\cdot\alpha^{2\cdot\gamma_n} \leq (n/\log^3(n))$ (by Lemma 4.4). Then, $x\cdot(y/m) \geq (n/\log^3(n))$; and by substitution into the formula of Lemma 4.3, the result is obtained. $\square$

LEMMA 4.11. *With probability greater than or equal to*

$$1 - \lceil(\log_{3/2}(n))\rceil(\frac{2}{e})^{n/\log^3(n)},$$

the number of edges that become introvert is less than $6.5 \cdot n$.

Proof. Assume that the bounds expected in Lemmas 4.8 and 4.10 are obtained. (The probability for this will be computed later.)

Lemma 4.8 is used to compute the number of edges that become *introvert* in the first iterations, and Lemma 4.10 is used to compute this number for the last iterations.

The sum is bounded by:

$$\left(\sum_{i=0}^{\gamma_n - 1} n \cdot \alpha^i\right) + \sum_{\gamma_n}^{\lceil \log_{3/2}(n)\rceil} \frac{n}{\log^2(n)}.$$

For a large enough n, the sum can be bounded by

$$\left(\sum_{i=0}^{\infty} n \cdot \alpha^i\right) + \frac{n}{\log^2(n)} \cdot \log^2(n) = n + n \cdot \sum_{i=0}^{\infty} \alpha^i = n + \frac{n}{1 - \sqrt{2/3}} < 6.5 \cdot n.$$

Now we compute an upper limit to the probability of failure. Failure can happen if:

1. The number of edges that become *introvert* in iteration $i < \gamma_n$ is more than $n \cdot \alpha^i$. By Lemma 4.8, the probability is bounded by $(2/e)^{n/\log^3(n)}$.
2. The number of edges that become *introvert* in iteration $i \geq \gamma_n$ is more than $\log^2(n)$. By Lemma 4.10, the probability is bounded by $(2/e)^{n/\log^3(n)}$.

An upper bound to the number of iterations ($\lceil \log_{3/2}(n)\rceil$) is given by Lemma 4.5. Let us assume that failure in any iteration causes the algorithm to fail. This assumption may be false, but it gives us an upper bound on the probability of failure. By adding up the probabilities of failure in all the iterations, we get an upper bound for the probability that the algorithm fails. Multiplying the upper bound for the number of iterations by the upper bound for the probability of failure in each iteration yields the claimed probability. □

THEOREM 4.12. *The time complexity of the dense-to-easy algorithm is* $O(\log(n))$.

Proof. The time complexity of iteration i is $O(1 + (m/P) \cdot \alpha^i)$. By Lemma 4.5, the number of iterations is $O(\log(n))$. Therefore, the complexity of the **for** loop is

$$O\left(\sum_{i=0}^{O(\log(n))} 1 + \frac{m}{P} \cdot \alpha^i\right) = O(\log(n)).$$ □

COROLLARY 4.13. *With probability greater than or equal to* $1 - \lceil \log_{3/2}(n)\rceil \cdot (2/e)^{n/\log^3(n)}$, *the running time of the connectivity algorithm for a dense graph is* $O(\log(n))$.

Proof. The time complexity of the dense-to-easy algorithm is $O(\log(n))$ (by Theorem 4.12). The probability of having more than $O(n)$ edges was bounded above by Lemma 4.11. In a dense graph, $O(n) \leq O(P)$; therefore, by Theorem 3.8, the easy-case algorithm will run in $O(\log(n))$ time. □

5. Sparse-to-dense reduction. The easy case offered one processor per vertex and per edge. The dense graph had too many edges for the number of processors, relative to the algorithm of Shiloach and Vishkin [11] so we found a way of getting a sparser graph. Clearly, we cannot use this approach if the graph is already sparse. In the sparse-graph case, we have only $o(n)$ processors, so we cannot have one per vertex.

Our solution transforms the input graph into one with few enough vertices that the dense-to-easy reduction can be used. We do this by a process of (repeatedly) partitioning the graph into a sparse part and a dense part. In the dense part, we combine adjacent vertices into supervertices and replace all edges between nonroots by edges adjoining the respective roots. We repeat the partitioning algorithm $2 \cdot \log(\log(n))$ times, always on the remaining sparse subgraph. At the end, we have a transformed graph that consists of the sparse subgraph that remains after the last iteration and the accumulated union of all of the dense subgraphs returned by *partitioning*. The size of this graph is $O(n/\log(n))$. This permits us one processor per vertex, and we proceed with the dense-to-easy reduction.

At each iteration we take a sample of edges from the sparse subgraph, starting with the complete set. When we join the vertices into supervertices we reduce the size of the remaining sparse subgraph by $\frac{1}{3}$. After $2 \cdot \log(\log(n))$ applications of *partitioning* this sparse subgraph has only $O(n/\log(n))$ vertices. Each iteration i of *partitioning* adds at most $O(n_i/\log(n))$ vertices to the dense subgraph. The number n_i decreases geometrically, so after $2 \cdot \log(\log(n))$ applications of *partitioning* on the sparse subgraph, the number of dense vertices is at most $\sum_{i=0}^{\infty}(\frac{2}{3})^i \cdot (n/\log(n)) = (n/\log(n))$.

The process by which vertices are combined into supervertices presents some problems. We must ensure fast ($O(1)$) transmission of information in any tree from leaves to root, and the Shiloach and Vishkin approach [11] can create trees too deep for this. We want trees of depth one, and present a deterministic algorithm to create them.

5.1. Deterministic mating. Shiloach and Vishkin's algorithm [11] creates trees with a height of $O(n)$. Cole and Vishkin [3] avoided this problem by using parallel union-find, a method that increases the processors' complexity of their connected components algorithm by an inverse Ackerman function. Reif [10], who introduced the term "mating" in his connectivity algorithm, solved the problem by using a randomized procedure *Random-Mate* that creates trees of height one. His solution is simple but has probability of $\frac{1}{2}$ of creating too few trees. The ideal procedure is one that will create trees of height one, such that most of the vertices will belong to those trees. A similar idea was proposed by Hagerup [7].

Our algorithm, *deterministic-mate*, uses an auxiliary data structure, the *next* graph. Each vertex v carries one additional pointer, $next(v)$ to some adjacent vertex. This allows us to classify vertices as follows:

1. V_0—These vertices have input degree zero.
2. V_1—These vertices have input degree of one.
3. V_2—These vertices have input degree of two or more.

Vertices in V_0 have only one adjacency and are therefore easy to handle.

The number of V_2 vertices is bounded, as we shall see, by the number of V_0 vertices. However, the V_1 vertices can be a problem as they can form chains that take $O(\log(n))$ to rank. We manage by running only the first iterations of a list-ranking algorithm.

Each processor selects a vertex v. If $next(v)$ was not also selected at this step, a

```
procedure deterministic-mate (V, k)
/* Every vertex v ∈ V has an edge e_v = (v, u). */
/* Every processor is responsible for at most k vertices (in a local array) */
for every vertex v with edge e_v = (v, u) do next(v) := u od
for every vertex v in parallel do
   if v has indegree 0, then parent(v) := next(v) fi
   if v has (in the next data structure) either
      1. indegree 0
      2. indegree 2 or more
      3. is a parent of a vertex of indegree 0
   then remove v from the graph and the array.
   fi
od
/* Ranking the lists */
for i := 1 to k + 2 · ⌈log(log(n))⌉ do
   every non-busy processor takes the next vertex v from its array.
   if next(v) was not taken in this step
      then parent(v) := parent(next(v)). Remove next(v).
   else
/* If not, then we have a "chain" of v, next(v), and possibly next(next(v)), ... */
      break the "chain" of vertices into sublists of size 2 to 2 · ⌈log(log(n))⌉.
      The processor which is responsible for the last element in a sublist s
      (no vertex in the sublist s points to it) collects the vertices in sublist s.
      for every vertex w in the sublist s do
         parent(w) :=last element of the sublist s.
      od
   fi
od

end deterministic-mate
```

FIG. 3. *Deterministic mating.*

jump-over is used: $parent(v)$ is set to $parent(next(v))$, and $next(v)$ is removed from the graph to shorten the list. The vertex v remains in the graph. It is used to establish the link between any descendants it may have in the *next* graph and the root of its supervertex (which is reached through parent pointers).

Of course, if a vertex, its *next*, the *next* of its *next*, and so on, happen to be chosen at the same iteration, we cannot use this method. Following Cole and Vishkin [3], we break the list into small sublists. The processor responsible for the last element z of some sublist s follows the pointers to pick up all the other vertices in s. This creates a star with root z and the rest of the vertices of s as leaves.

Our procedure is based on Anderson and Miller's deterministic list ranking [1]. (A similar algorithm was developed earlier by Cole and Vishkin [3].) The outline for our joining procedure, *deterministic-mate*, is presented in Fig. 3.

DEFINITION 5.1. *$V_{0,2}$ is the subset of vertices with indegree 0, 2, or parents of vertices with indegree 0.*

Every vertex points to some other vertex, with its *next* pointer. Vertices with indegree 0 are easily managed, as there is no choice to make. We attach them to their *next*'s and make stars from them. The following lemmas establish our claim about the number of stars.

LEMMA 5.2. *The number of vertices with indegree 0 is greater than or equal to the number of vertices with indegree 2 or more.*

Proof. We argue the pigeon-hole principle. There are as many pointers as vertices.

If some vertex is pointed to by more than one pointer, then some other vertex is not pointed to at all. □

LEMMA 5.3. *After the application of deterministic-mate, the number of stars composed of vertices of $V_{0,2}$ is at most $\frac{2}{3} \cdot |V_{0,2}|$.*

Proof. Every vertex of indegree 0 becomes part of some star, but not the root. The number of roots, and therefore stars, is bounded by the number of vertices of indegree two or more and the number of vertices having a child of indegree 0. By Lemma 5.2, the first number is bounded by the number of vertices of indegree 0. Since each vertex has only one pointer, so, clearly, is the second.

Therefore the number of roots is at most $\frac{2}{3} \cdot |V_{0,2}|$. □

Once we have dealt with these stars, what remains are the V_1 vertices. Some of these have indegree 0 (if a child was removed). This leaves a simple structure to which we can apply algorithms similar to those used for standard list ranking ([3], [1]).

Ideally, every processor would pick a vertex v and "try" to create a star of two vertices, v and $next(v)$. However, it might happen that a second processor, in the same time slot, has chosen to create a star of $next(v)$ and $next(next(v))$. This could result in a long list $(v, next(v), next(next(v)) \cdots)$. We want to break this list into smaller sublists, and let one processor take care of each sublist.

LEMMA 5.4. *We can break our structure into sublists such that no sublist is shorter than 2 or longer than $2 \cdot \lceil \log(\log(n)) \rceil$ in constant time using an optimal number of processors.*

Proof. This result was proved by Cole and Vishkin in [4]. We present here an outline of their proof. Assume that a vertex j is the *next* of a vertex i. Define $SERIAL_1(i)$ as "the index of the rightmost bit in which i and j differ." If $SERIAL_1(i) = SERIAL_1(j) = k$, then the kth bit in i is different from the kth bit in j. We say that $SERIAL_1(i) > SERIAL_1(j)$ if either $SERIAL_1(i)$ is greater than $SERIAL_1(j)$ or they are equal, and the rightmost bit in which i and j differ is 1 in i.

Note that:

- $SERIAL_1(i)$ is at most $\lceil \log(n) \rceil$ for $i, next(i) \leq n$.
- For every i, $SERIAL_1(i)$ and $SERIAL_1(next(i))$ cannot both be local maxima in the *next* list.
- Every $2 \cdot \lceil \log(n) \rceil$ (or fewer) elements there is a local maximum of the $SERIAL_1$ function in the list.

Breaking the list into sublists in the locations of the local maxima results in sublists of length 2 to $2 \cdot \lceil \log(n) \rceil$. Applying the $SERIAL$ function to the sublists, this time to the $SERIAL_1$ numbers, yields $SERIAL_2$ numbers.

By breaking each sublist at the local maxima of the $SERIAL_2$ function, we get the resulting sublists as claimed. □

Every small list can be scanned by one processor in time complexity equal to the length of list.

LEMMA 5.5. *Every vertex v that was scanned in the first k iterations of the list-ranking part of the deterministic-mate algorithm, will belong to a tree of height* 1.

Proof. If v was removed, then it became a root for the vertex that pointed to v through the *next* pointer (these two vertices create a star with two vertices). If it was not removed, then it became either a leaf or part of a list. If it was a leaf, then it gets a parent. If it was part of a list, then the list is ranked in $2 \cdot \log(\log(n))$ iterations (or less), and the vertex becomes a part of a star that represents the elements of the list. □

LEMMA 5.6. *The number of vertices that were not processed in the first k iterations is bounded by the number of vertices that became leaves in the list ranking part.*

Proof. The only reason that a vertex was not processed is because its processor was busy compressing lists. Every vertex in a list can cause only one vertex not to be processed. □

THEOREM 5.7. *The number of stars in the resulting graph of the deterministic-mate algorithm is at most* $\frac{2}{3} \cdot |V|$.

Proof. Every vertex either belongs to $V_{0,2}$ or $V - V_{0,2}$. By Lemma 5.3, the number of stars composed of $V_{0,2}$ vertices is at most $\frac{2}{3}$ of $V_{0,2}$.

Every vertex v in $V - V_{0,2}$ either became the root of a star or a leaf of a star or was not processed at all by its processor. Every vertex that became a leaf prevented one vertex (its parent) from becoming a leaf. By Lemma 5.6, the number of vertices that were not processed is at most the number of vertices that became leaves as a result of performing the list-ranking part of the algorithm.

Therefore, every leaf can cause at most two vertices to be roots, and the proof follows. □

THEOREM 5.8. *The time complexity of the mating algorithm is* $O(k+\log(\log(n)))$.

Proof. The **for** loop is executed $k + 2 \cdot \lceil \log(\log(n)) \rceil$ times, and each iteration takes constant time. □

5.2. Partitioning the vertices. The algorithm given in Fig. 4 takes a graph $G(V_i, E_i)$ and a number n as input and creates two sets of supervertices. Each supervertex is a star and every edge (u, v) is replaced by edge $(root(u), root(v))$. The advantage of this organization is that the nonroot vertices have no live edges and therefore there is no need to deal with them. The two sets of supervertices are:

1. *extrovert*—A set of *extrovert* supervertices, each of which had at least one live incident edge in every edge sample that we took throughout the execution of the *partitioning* algorithm.
2. *introvert*—all the other supervertices.

We "mate" *extrovert* supervertices using the procedure *deterministic-mate* and reduce the number of *extrovert* supervertices (roots) by a factor of at least $\frac{1}{3}$. After $2 \cdot \log(\log(n))$ iterations, the number is reduced by a factor of at least

$$\left(\frac{2}{3}\right)^{2\cdot\log(\log(n))} < (1/\log(n));$$

so the number of *extrovert* supervertices after the *partitioning* algorithm is at most $|V_i|/\log(n)$.

In the *partitioning* algorithm we try to achieve three goals:

1. Reducing the number of vertices in the graph by a constant factor.
2. Reducing the number of edges in the *introvert* graph to $O(|V_i|)$.
3. Setting aside a set *extrovert* that is smaller than the input by a factor of $\log(n)$.

The first and the second goals are important because the time complexity of the procedure is proportional to its input size. The third goal is important because we want to reduce the number of vertices in the graph and therefore we want only a small set.

The algorithm uses ideas similar to those of the *dense-to-easy* algorithm in §4. We take a sample of edges and use it to join *extrovert* vertices. Using arguments similar

to those we used in §4 we prove that we achieve the second and the third goals. The first goal is achieved because the first sample includes all the edges of the graph and therefore the number of live supervertices in the graph is reduced by $\frac{1}{3}$.

The edges, whether live or dead, are stored in an array of length m. By keeping the edges in blocks of equal size and assigning a processor to each block, a sample of the edges (for the *partitioning* algorithm) can be picked in time proportional to sample size$/P$. This way, a sample can be chosen without repetition. Note that to simplify parts of the analysis, choosing with repetition is assumed; this is safe, for it only decreases the probability of an edge being chosen.

By taking a geometrically decreasing sample of edges, the probability of a supervertex not having an edge in the sample increases geometrically. However, the number of *extrovert* supervertices for each iteration decreases geometrically and the sample size decreases more slowly than the number of *extrovert* supervertices.

Therefore the number of live edges connected to vertices in *introvert* is proportional to the number of vertices in the input set.

We do not have enough processors to reduce the height of the trees. Their height may increase by at most 1 every time that we perform *deterministic-mate* because we always link roots.

We do perform a height-reduction operation once after the vertices are partitioned. We do it in the reverse order of the joining process. We assume that every vertex v that was a root in some iteration j is either a root or a child of some root. The children of v from iteration $j-1$ become children of $root(v)$ after one jump over. After all these trees become stars every live edge (u, v) in the graph is replaced by edge $(root(u), root(v))$ in $O(|E_i|/P)$ time.

5.3. Analysis of the partitioning algorithm.

DEFINITIONS:

1. An *extrovert root* r is a root with a value of *flag* $> j$ at the end of the jth iteration in loop j. At the start of the *partitioning* algorithm, every root is *extrovert.*
2. An *extrovert supervertex* v is a supervertex whose root is *extrovert.*
3. An *extrovert vertex* v is a vertex that belongs to a tree whose root is *extrovert.*
4. An *introvert supervertex* v is a supervertex whose root is not *extrovert.*
5. An *introvert vertex* is a vertex that belongs to a tree whose root is not an *extrovert.*
6. An *extrovert edge* is a live edge that connects two *extrovert* vertices.
7. An *extrovert edge* becomes an *introvert* edge if one of its end supervertices has become an *introvert.*
8. We say that a vertex or supervertex is *introvert-isolated* if it is in *introvert* and has no live edge incident on any other supervertex in *introvert.*
9. V_i and E_i are input variables to the partition procedure. V_i is a set of vertices and E_i a set of edges; $n_i = |V_i|$ and $m_i = |E_i|$ are the sizes of the input vertices and edges sets, respectively.

LEMMA 5.9. *The number of* introvert *vertices that are not isolated with respect to E_i after the* partitioning *algorithm stops is at most $n_i \cdot \alpha^2$.*

Proof. In the first iteration we pick a sample of size m_i. Therefore every nonisolated vertex has at least one edge in the sample. By Theorem 5.7 the number of vertices after the *deterministic-mate* algorithm is at most $\frac{2}{3}$ of the vertices that are mated. This number is an upper bound on the size of nonisolated *introvert* vertices. ∎

```
procedure partitioning (V_i:set-of-vertices; E_i:set-of-edges; n:integer)
                    returns (extrovert, introvert)
for every vertex v ∈ V_i in parallel do
   parent(v) := v
   flag_v := 0
od
for j := 0 to ⌈2 · log(log(n))⌉ do /* loop j */
   Pick a sample of edges of size max(P, ⌈|E_i| · α^j⌉)
   if (v, u) is a live edge in the sample and flag_root(u) = flag_root(v) = j then
     using concurrent write in parallel do
        e_root(v) := (v, u); e_root(u) := (u, v); flag_root(u) := j + 1; flag_root(v) := j + 1
     od
   fi
   k := ⌈⌈|E_i| · α^j⌉ / P⌉
   V := the set of roots v such that flag_v = j + 1
   If a processor wrote the edge e_v then it puts v into its local array.
   call deterministic-mate(V, k)
od /* loop j */
 for j := ⌈2 · log(log(n))⌉ − 1 downto 0 do
    for every v ∈ V_i in parallel do
       if v was mated in the jth iteration of loop j
         then parent(v) := parent(parent(v))
       fi
    od
od
for every edge (u, v) in the graph in parallel do
   replace edge (u, v) by edge (root(u), root(v))
od
for every vertex v ∈ V_i in parallel do
   if flag_v = ⌈2 · log(log(n))⌉ + 1 then mark v as extrovert.
     else mark v as introvert.
   fi
od
return (extrovert, introvert)

end partitioning
```

FIG. 4. *Algorithm for partitioning of the vertices.*

LEMMA 5.10. *The number of extrovert supervertices in iteration k of loop j is at most $n_i \cdot \alpha^{2 \cdot k}$.*

Proof. The proof follows immediately from Theorem 5.7 because every *extrovert* root has an edge in the sample. □

LEMMA 5.11. *For a given x, the probability that at least x edges will become introvert in iteration k of loop j is bounded by*

$$e^{-x \cdot (y/m_i)} \cdot 2^{n_i \cdot \alpha^{2 \cdot k}},$$

where $y = \max(P, \lceil m_i \cdot \alpha^k \rceil)$.

Proof. In every iteration a subset of (possibly empty) vertices becomes *introvert.* By Lemma 4.2, the probability that a particular subset of vertices with x *extrovert* edges will become *introvert* is at most $e^{-x \cdot (y/m_i)}$. By Lemma 5.10, we have at most $n_i \cdot \alpha^{2 \cdot k}$ *extrovert* supervertices, and therefore we have at most $2^{n_i \cdot \alpha^{2 \cdot k}}$ subsets with x *extrovert* edges.

We find an upper bound to the probability that at least one such subset will become *introvert* by summing up the probabilities of all these events. Therefore, an upper limit to the probability is $e^{-x\cdot(y/m_i)} \cdot 2^{n_i\cdot\alpha^{2\cdot k}}$. □

Please note that the bound is not tight for reasons similar to those that explain the looseness of the bound in Lemma 4.3.

DEFINITION 5.12. Let f_k be the probability that the number of *extrovert* edges that become *introvert* during iteration k of loop j is greater than $n_i \cdot \alpha^k$.

LEMMA 5.13. *The probability f_k is bounded by $(2/e)^{n_i\cdot\alpha^{2\cdot k}}$.*

Proof. Set $x = n_i \cdot \alpha^k$, $z = m_i \cdot \alpha^k$; by Lemma 5.10, the number of supervertices is bounded by $n_i \cdot \alpha^{2\cdot k}$. Note that $y \geq z$. Then, by Lemma 5.11,

$$f_k \leq e^{-n_i\cdot\alpha^k\cdot\alpha^k} \cdot 2^{n_i\cdot\alpha^{2\cdot k}} = \left(\frac{2}{e}\right)^{n_i\cdot\alpha^{2\cdot k}}. \qquad \square$$

THEOREM 5.14. *With probability greater than or equal to*

$$1 - (\lceil 2 \cdot \log(\log(n)) \rceil) \cdot \left(\frac{2}{e}\right)^{n_i/\log^2(n)}$$

the number of edges that became introvert *is less than* $5.5 \cdot |n_i|$.

Proof. Assume that the bound expected in Lemma 5.13 is obtained. (The probability for this will be computed later.)

The number of edges that become *introvert* is bounded by:

$$\left(\sum_{k=0}^{2\cdot\log(\log(n))} n_i \cdot \alpha^k\right) < 5.5 \cdot n_i.$$

Now we compute an upper limit to the probability of failure. Failure can happen if the number of edges that become *introvert* in some iteration k of loop j is more than $n_i \cdot \alpha^k$. By Lemma 5.13, the probability is bounded by

$$\left(\frac{2}{e}\right)^{n_i\cdot\alpha^{4\cdot\log(\log(n))}} < \left(\frac{2}{e}\right)^{n_i/\log^2(n)}.$$

Let us assume that failure in any iteration causes the algorithm to fail. This assumption may be false, but it gives us an upper bound on the probability of failure. By adding up the probabilities of failure in all the iterations, we get an upper bound for the probability that the algorithm fails. Multiplying the upper bound for the number of iterations by the upper bound for the probability of failure in each iteration yields the claimed probability. □

Note that the number of iterations is $2\cdot\log(\log(n))+1$, but there can be no failure in the first iteration because every live edge is in the sample.

THEOREM 5.15. *The time complexity of the partitioning algorithm is*

$$O\left(\frac{m_i}{P} + (\log(\log(n)))^2\right).$$

Proof. Every iteration k in loop j is composed of

1. Taking a sample of edges and writing every edge (u, v) to *root* (u) and *root* (v).

procedure *sparse-to-dense* $(G(V,E))$
Extrovert-Set$:= \emptyset$; $V_{-1} := V$; $E_{-1} := E$; $n := |V|$

for $i := 0$ **to** $\lceil 2 \cdot \log(\log(n)) \rceil$ **do**
  $(extrovert, introvert) := partitioning(V_{i-1}, E_{i-1}, n)$
  Extrovert-Set := *Extrovert-Set* $\cup$ *extrovert*
  $V_i :=$ vertices in *introvert* which are not *introvert-isolated*
  $E_i := \{(v,u) | (v,u)$ is a live edge, and $u, v \in introvert\}$
  Redistribute V_i and E_i evenly among the processors.
od
for every root $v \in introvert$ **in parallel do**
  if there exists an *extrovert* vertex u adjacent to v, **then**
    $parent(v) := u$; replace every edge (v,w) by edge (u,w).
  fi
od
$V :=$ *introvert* $\cup$ *Extrovert-Set*; $E :=$ all live edges.
return $G(V,E)$

end *sparse-to-dense*

FIG. 5. *Reducing the number of vertices.*

2. Mating the *extrovert* trees using the *deterministic-mate* procedure.

The time complexity of the first term is bounded by the product of the sample size and maximum height of the supervertex trees in that iteration (because we have to write every edge in the sample to the roots). The sample size per processor is bounded by $1 + \alpha^k \cdot (m_i/P)$, and the height of every tree is at most k. Therefore the amount of work executed by every processor is bounded by $k \cdot (1 + \alpha^k \cdot (m_i/P))$.

By Theorem 5.8, the time complexity for the second term is

$$O(\log(\log(n)) + \alpha^k \cdot (m_i/P)).$$

Thus the time complexity for the *partitioning* is:

$$\sum_{k=0}^{\lceil 2\cdot\log(\log(n))\rceil} 1 + (1+k) \cdot \left(\frac{m_i}{P} \cdot \alpha^k\right) + \log(\log(n)) + \frac{m_i}{P} \cdot \alpha^k$$
$$\le O\left((\log(\log(n)))^2 + \sum_{k=0}^{\infty} (k+1) \cdot \frac{m_i}{P} \cdot \alpha^k \right) = O\left((\log(\log(n)))^2 + \frac{m_i}{P} \right). \quad \square$$

THEOREM 5.16. *After performing the algorithm, the size of extrovert is at most* $n_i/\log(n)$.

Proof. The proof follows immediately from Lemma 5.10. $\square$

5.4. The algorithm for reducing the number of vertices. The algorithm presented in Fig. 5 reduces the number of vertices in the graph from n to $n/\log(n)$. The expected time complexity of the algorithm is $O((n+m)/P + (\log(\log(n)))^3)$.

Our goal is to reduce the number of supervertices in the graph. We call *partitioning* $2 \cdot \log(\log(n))$. Each call returns *extrovert* and *introvert* sets. We set the *extrovert* supervertices aside, and call *partitioning* again with the *introvert* set as the input.

After the ith call to *partitioning*, the number of new *extrovert* supervertices is $|V_i|/\log(n)$ (by Theorem 5.16). Since by Lemma 5.9, the size of *introvert* decreases

geometrically, there are $O(n/\log(n))$ supervertices in *Extrovert-Set* after performing the loop.

The size of *introvert* ($|V_i|$) is reduced by a constant factor in each call of *partitioning* and therefore after $O(\log(\log(n)))$, its size is reduced by a factor of $\log(n)$ from the original size of the vertex set.

The number of *introvert* edges is also reduced in every call to *partitioning* (see Theorem 5.14), and therefore the complexity of each iteration is reduced by a constant factor.

5.5. Analysis of the algorithm that reduces the number of vertices.

LEMMA 5.17. *After iteration i, the number of live supervertices that are not* introvert-isolated *is bounded by $n \cdot \alpha^{2 \cdot i}$.*

Proof. The proof follows immediately from Lemma 5.9. □

LEMMA 5.18. *With probability greater than or equal to*

$$1 - (\lceil 2 \cdot \log(\log(n)) \rceil) \left(\frac{2}{e}\right)^{|V_i|/\log^2(n)},$$

the number of edges that become introvert by the end of each iteration i is less than $5.5 \cdot |V_i|$.

Proof. The proof follows immediately from Theorem 5.14. □

Note that by definition, *introvert-isolated* vertices have no edges incident to other supervertices in *introvert.* Therefore the number of these vertices has no influence on the number of edges that will become *introvert* in the next iteration.

LEMMA 5.19. *With probability greater than or equal to*

$$1 - (\lceil 2 \cdot \log(\log(n)) \rceil) \left(\frac{2}{e}\right)^{n/\log^3(n)},$$

the number of live edges in E_i is bounded by $5.5 \cdot \max(n \cdot \alpha^{2 \cdot (i-1)}, n/\log(n))$.

Proof. If $V_i \geq (n/\log(n))$, then the proof follows immediately from Lemma 5.17 and Theorem 5.14.

If V_i is smaller than $n/\log(n)$, the claim is true because we can, strictly for purposes of analysis, add vertices with degree zero to V_i. They will not affect the actual running of the algorithm, but we will be able to use Theorem 5.14 for that case as well. □

THEOREM 5.20. *With probability,*

$$1 - 4 \cdot (\lceil \log(\log(n)) \rceil)^2 \cdot \left(\frac{2}{e}\right)^{n/\log^3(n)},$$

the time complexity of the algorithm is $O(\log(n))$.

Proof. The time complexity of the *partitioning* is $O((|E_i|/P) + (\log(\log(n)))^2)$ (by Theorem 5.15). During the algorithm, for every edge for which it is responsible, a processor can process the edge (that is, read information, update its status, etc.) in constant time.

Our goal is for each processor to have about the same number of edges of E_i. That is, in iteration i of the algorithm, if E_i is the set of edges remaining for the next

partitioning, then each processor should have $O(|E_i|/P)$ edges in order to balance the work.

We use a technique developed by Cole and Vishkin [5] for their connected components algorithm [3]. They gave an optimal algorithm to compute the prefix sum of an array of size m in $O(\log(m)/\log(\log(m)))$ time using an optimal number of processors.

The load balancing is done in the following way:

1. Every processor computes the number of live edges it has.
2. A PRAM array A of size P is allocated; every processor p_i writes the number of its live *introvert* edges in $A[i]$.
3. The fast prefix sum algorithm is used on array A.
4. A PRAM array B of the size of the number of live *introvert* edges is allocated.
5. Every processor i writes its edges in array B in the interval $[A[i-1]+1, A[i]]$.
6. B is divided into equal parts between the processors.

The first *partitioning* takes $O(m/P)$ time. Therefore, the expected time complexity is bounded by

$$\frac{m}{P} + \sum_{i=0}^{\lceil 2\cdot\log(\log(n))\rceil} \left(\frac{n}{P}\cdot\left(\frac{2}{3}\right)^{i} + (\log(\log(n)))^2 + \frac{\log(m)}{\log(\log(m))}\right)$$

$$\leq O\left(\frac{m}{P} + \log(m) + \frac{n}{P}\right) = O(\log(n)).$$

We assume that if, in any iteration of the *partitioning* algorithm, more than the expected number of live edges became *introvert*, then the algorithm failed. The *sparse-to-dense* algorithm calls the *partitioning* algorithm $\lceil 2\cdot\log(\log(n))\rceil$ times, and every time the *partitioning* algorithm runs for $\lceil 2\cdot\log(\log(n))\rceil$ iterations.

If $V_i \leq (n/\log(n))$, and all the previous iterations have succeeded, then every processor has a constant number of edges and vertices, and every iteration takes $O((\log(\log(n)))^2)$ plus load-balancing time.

If $V_i > (n/\log(n))$, then by Theorem 5.14 the probability of failure in any call to *partitioning* is bounded by

$$\lceil 2\cdot\log(\log(n))\rceil \cdot \left(\frac{2}{e}\right)^{n_i/\log^2(n)}$$

We call *partitioning* $\lceil 2\cdot\log(\log(n))\rceil$ times, and an upper bound for the probability of failure is the number of calls times the probability to fail at each call. □

THEOREM 5.21. *The number of live supervertices at the end of the algorithm is* $O(\frac{n}{\log(n)})$.

Proof. The number of *introvert* supervertices is $O(\frac{n}{\log(n)})$ (Lemma 5.17). The set of supervertices that moves to the *Extrovert-Set* after iteration i is bounded by $n_i/\log(n)$ (Theorem 5.16), and n_i decreases geometrically (Lemma 5.17). □

Note that we do not know the number of edges after the algorithm stops. It is possible that the *extrovert* supervertices have most of the edges of the original graph, in which case we need the *dense-to-easy* procedure to reduce their number.

6. Algorithm for finding connected components. The algorithm presented in Fig. 6 takes a graph $G(V, E)$ as input, and finds its connected components. After

execution, every vertex should point to the root of the connected component supervertex to which it belongs. However, the leaves of stars created by the *sparse-to-dense* and *dense-to-easy* procedures point to arbitrary dense vertices, rather than specifically to the root of the appropriate supervertex. We fix this problem after completing the *easy-case* algorithm by using a double jump-over operation, that is, making each vertex point to its great-grandparent.

The connected components algorithm is a *Las-Vegas* algorithm in the sense that it always gives the right answer but may take longer than expected.

procedure *general-graph* $(G(V,E))$
  call *sparse-to-dense*(G)
  call *dense-to-easy*(G)
  call *easy-case*(G)
  for every $v \in V$ **in parallel do**
    $parent(v) := parent(parent(parent(v)))$
  od
end *general-graph*

FIG. 6. *Finding connected components in a graph.*

6.1. Analysis of the algorithm for finding connected components.

THEOREM 6.1. *The expected time complexity of the algorithm is* $O(\log(n))$.

Proof. The expected complexity of the sparse-to-dense procedure is $O(\log(n))$ by Theorem 5.20. If the procedure succeeds, then by Theorem 5.21, we have a constant number of vertices per processor. In that case we can run procedure dense-to-easy in $O(\log(n))$ time (see Theorem 4.12). If dense-to-easy succeeds, then we can run Shiloach and Vishkin's algorithm in $O(\log(n))$ time [11]. The total complexity is: $O(\log(n)) + O(\log(n)) + O(\log(n)) = O(\log(n))$. □

THEOREM 6.2. *The probability that the algorithm will run longer than expected is bounded by*

$$\left(4 \cdot \left(\lceil \log(\log(n)) \rceil\right)^2 + \lceil \log_{\frac{3}{2}}(n) \rceil\right) \cdot \left(\frac{2}{e}\right)^{n/\log^3(n)}.$$

Proof. The proof follows from Lemma 4.11 and Theorem 5.20. □

Acknowledgment. I am grateful to Gary L. Miller for his advice and help in the preparation of this paper, and to the referees for many insightful suggestions and corrections. I would also like to thank Salit Gazit and Mara Chibnik for their help in editing the paper.

REFERENCES

[1] R. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in Proc. Aegean Workshop on Computing, 1988, Corfu, Greece, pp. 81–90.

[2] D. H. A. CHANDRA AND D. V. SARWATE, *Computing connected components on parallel computers*, Comm. ACM, 22 (1979), pp. 461–464.

[3] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list tree, and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, October 1986, Toronto, Ontario, Canada, pp. 478–491.

[4] ———, *Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 206–219.

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

[6] H. GAZIT, *An optimal randomized parallel algorithm for finding connected components in a graph*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, October 1986, Toronto, Ontario, Canada, pp. 492–501.

[7] T. HAGERUP, *Optimal parallel algorithms on planar graphs*, in Proc. Aegean Workshop on Computing, Corfu, Greece, 1988, pp. 24–32.

[8] A. KANEVSKY AND V. RAMACHANDRAN, *Improved algorithms for graph four-connectivity*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, October 1987, pp. 252–259.

[9] G. L. MILLER AND V. RAMACHANDRAN, *A new graph triconnectivity algorithm and its parallelization*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, May 1987, pp. 335–344.

[10] J. H. REIF, *Optimal parallel algorithms for graph connectivity*, Tech. Report TR-08-84, Center for Computing Research, Harvard University, Cambridge, MA, 1984.

[11] Y. SHILOACH AND U. VISHKIN, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[12] R. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–874.

[13] J. C. WYLLIE, *The complexity of parallel computation*, Tech. Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1068–1083, December 1991
© 1991 Society for Industrial and Applied Mathematics
006

ASYMPTOTICALLY FAST TRIANGULARIZATION OF MATRICES OVER RINGS*

JAMES L. HAFNER† AND KEVIN S. MCCURLEY†‡

Abstract. This paper considers the problems of triangularizing and diagonalizing matrices over rings, with particular emphasis on the integral case. It begins with a description of fast algorithms for the computation of Hermite and Smith normal forms of integer matrices. Then it shows how to apply fast matrix multiplication techniques to the problem of triangularizing a matrix over a ring using elementary column operations. These general results lead to an algorithm for triangularizing integer matrices that has a faster running time than the known Hermite normal form algorithms. The triangular matrix that is computed has small entries like the Hermite normal form, and will suffice for many applications.

Key words. Hermite normal form, factorization of matrices, algorithms

AMS(MOS) subject classifications. 68Q40, 11Y16, 15A23

1. Introduction. A common theme when performing computations involving matrices is to use a triangular or diagonal matrix that is equivalent for the computational task at hand. For example, Hermite and Smith normal forms of integer matrices (see §2 for definitions) have applications to a number of areas including solving systems of linear Diophantine equations (see [13]), integer programming (see [28]), algorithmic problems in lattices (see [14] for an example), and the structure theorem for finitely generated Abelian groups (see [15, Chap. 7–10]).

In this paper, we consider the problem of computing such special forms of matrices over rings, with particular emphasis on the integers. In §2 we give deterministic algorithms for computing the Hermite and Smith normal forms of integer matrices which have fast and rigorously proved running times. While the emphasis of our paper is on the asymptotic analysis of algorithms, we believe these algorithms are quite practical.

The main results of this paper appear in §§3 and 4. In §3, we consider the problem of matrix triangularization over very general rings, employing the techniques of fast matrix multiplication. In §4, these methods are then applied to the special case of matrices over the integers to reduce the computational complexity of triangularization. Note that in many applications, a triangular matrix can be used in place of the Hermite normal form. In the last section, we discuss some open problems.

We, of course, require some notation. If A is a matrix over a commutative ring R, we let $\mathcal{L}(A)$ denote the R-module consisting of all R-linear combinations of the columns of A. In the case $R = \mathbb{Z}$, $\mathcal{L}(A)$ is the lattice generated by the columns of A. In this case, we let $\mathrm{Det}(\mathcal{L}(A))$ denote the determinant of this lattice. We will say that A is right unimodularly equivalent over R to a matrix B if there exists a unimodular matrix K such that $AK = B$. (A unimodular matrix K satisfies $\det(K) = \pm 1$.) Finally, we use the notation log to denote logarithms to base 2.

2. Hermite and Smith normal forms. Let A denote an $m \times n$ matrix with integer entries. A classical result says that if A has rank m, then there exists an $m \times n$

* Received by the editors January 16, 1990; accepted for publication (in revised form) January 11, 1991. An extended abstract of this work appeared in the Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

† IBM Almaden Research Center, K53/802, 650 Harry Road, San Jose, California 95120.

‡ Present address, Division 1423, Sandia National Laboratories, Albuquerque, New Mexico 87185.

lower triangular matrix H and an $n \times n$ unimodular matrix K such that $A = HK$, and

$$\begin{aligned} h_{ii} &> 0, && 1 \le i \le m, \\ 0 \le h_{ij} &< h_{ii}, && 1 \le i \le m, 1 \le j < i. \end{aligned}$$

This was first proved for the case of square matrices by Hermite [17]. The matrix H is unique, and is known as the Hermite normal form of A. Note that $\mathrm{Det}(\mathcal{L}(A)) = \prod_{i=1}^{m} h_{ii}$.

A related construction is the Smith normal form. Smith [29] proved that for every $m \times n$ matrix A of rank r, there exist unimodular matrices J, K and a diagonal matrix S with $A = JSK$, $S = \mathrm{diag}(s_1, \cdots, s_r, 0, \cdots, 0)$, and $s_1 \mid s_2 \mid \cdots \mid s_r$, with $s_i > 0$. The matrix S is unique, and is known as the Smith normal form of A. The integers $s_1, s_2, \cdots, s_r$ are called the invariant factors of A. If A has rank m, then the product of the invariant factors equals $\mathrm{Det}(\mathcal{L}(A))$.

In this section we shall describe some algorithms for computing the Hermite and Smith normal forms of integer matrices, and analyze the performance of the algorithms. Numerous researchers have previously given algorithms for the calculation of Hermite and Smith normal forms, as well as the related problem of solving systems of linear Diophantine equations. These include Hermite [17]; Smith [29]; Blankinship [2], [3]; Bradley [5]; Havas and Sterling [16]; Frumkin [12], [11]; Domich [9]; Kannan and Bachem [23]; Domich, Kannan, and Trotter [10]; Kaminski and Paz [22]; Hu [18, App. A]; Schrijver [28, Chaps. 4–5]; McClellan [24]; Chou and Collins [7]; and Iliopolous [20].

The simplest algorithms for computing the Hermite normal form use a procedure similar to the Euclidean algorithm, replacing the diagonal element in a given row by the gcd of the elements in the same row to the right. This procedure takes $O(m^2 n)$ operations on rational numbers, but such algorithms are well known to suffer from "intermediate expression swell," since the numbers that are encountered during the calculation can become enormous, as was observed in [2] and [12], among others. As an example of the kind of intermediate expression swell that can occur, the first author used the software package Mathematica to run some trials on random 20×20 matrices with integer entries between 0 and 10. After reduction to triangular form using the standard Hermite procedure, they almost always gave at least one entry exceeding 10^{500}, and after several such examples, the following rather spectacular example was found. Consider the matrix

$$\begin{bmatrix}
10 & 8 & 10 & 5 & 5 & 10 & 0 & 8 & 5 & 9 & 8 & 3 & 0 & 9 & 9 & 2 & 7 & 6 & 0 & 4 \\
4 & 10 & 8 & 5 & 2 & 6 & 5 & 1 & 2 & 6 & 9 & 1 & 4 & 6 & 2 & 7 & 7 & 1 & 4 & 10 \\
2 & 4 & 6 & 4 & 3 & 0 & 3 & 0 & 10 & 2 & 3 & 0 & 6 & 7 & 5 & 7 & 7 & 3 & 3 & 2 \\
1 & 5 & 0 & 4 & 8 & 0 & 10 & 10 & 2 & 0 & 5 & 3 & 6 & 10 & 2 & 10 & 2 & 10 & 10 & 2 \\
9 & 8 & 5 & 4 & 10 & 6 & 3 & 10 & 3 & 8 & 7 & 4 & 7 & 1 & 5 & 8 & 6 & 8 & 3 & 1 \\
10 & 3 & 1 & 0 & 0 & 9 & 8 & 2 & 1 & 6 & 0 & 10 & 7 & 4 & 9 & 2 & 3 & 10 & 2 & 9 \\
8 & 2 & 8 & 0 & 0 & 9 & 3 & 0 & 7 & 8 & 0 & 3 & 4 & 5 & 3 & 8 & 1 & 4 & 4 & 9 \\
0 & 4 & 8 & 6 & 0 & 1 & 5 & 0 & 0 & 3 & 4 & 6 & 9 & 5 & 10 & 10 & 3 & 5 & 9 & 9 \\
4 & 6 & 4 & 8 & 9 & 2 & 5 & 6 & 9 & 8 & 9 & 3 & 6 & 7 & 3 & 8 & 5 & 10 & 1 & 7 \\
2 & 0 & 7 & 10 & 8 & 4 & 7 & 5 & 3 & 7 & 1 & 10 & 6 & 2 & 2 & 9 & 9 & 9 & 1 & 10 \\
2 & 2 & 1 & 5 & 4 & 1 & 4 & 4 & 6 & 7 & 0 & 10 & 0 & 2 & 5 & 1 & 9 & 10 & 2 & 5 \\
4 & 1 & 10 & 2 & 4 & 0 & 9 & 7 & 8 & 9 & 9 & 3 & 3 & 9 & 10 & 1 & 7 & 1 & 3 & 1 \\
5 & 3 & 3 & 9 & 2 & 1 & 4 & 2 & 7 & 8 & 5 & 7 & 5 & 5 & 9 & 4 & 5 & 4 & 10 & 6 \\
4 & 8 & 10 & 1 & 0 & 8 & 10 & 4 & 4 & 10 & 3 & 10 & 7 & 4 & 5 & 7 & 3 & 2 & 1 & 2 \\
6 & 9 & 4 & 7 & 10 & 3 & 6 & 9 & 5 & 8 & 4 & 6 & 9 & 8 & 7 & 5 & 9 & 8 & 9 & 7 \\
5 & 10 & 5 & 6 & 10 & 0 & 9 & 5 & 1 & 4 & 0 & 9 & 8 & 9 & 3 & 5 & 2 & 2 & 0 & 8 \\
1 & 3 & 10 & 2 & 5 & 5 & 1 & 1 & 10 & 6 & 10 & 4 & 3 & 8 & 1 & 6 & 1 & 2 & 6 & 9 \\
6 & 4 & 1 & 5 & 8 & 2 & 3 & 1 & 3 & 7 & 4 & 10 & 2 & 9 & 10 & 3 & 3 & 6 & 5 & 8 \\
3 & 7 & 5 & 2 & 10 & 0 & 10 & 4 & 0 & 0 & 10 & 3 & 2 & 9 & 5 & 1 & 7 & 10 & 5 & 3 \\
10 & 2 & 10 & 6 & 2 & 9 & 3 & 2 & 0 & 2 & 5 & 3 & 3 & 1 & 5 & 5 & 10 & 0 & 7 & 3
\end{bmatrix}$$

After reducing this matrix to triangular form, there is an entry exceeding 10^{5011}. By contrast, the Hadamard bound for the determinant of this matrix is approximately 10^{33}, and the algorithm that we present here would only require modular arithmetic on integers of this size. It should be mentioned, however, that the bad reputation of the classical Hermite algorithm is based on computational experience and heuristic arguments, and that it remains an open question whether the length of the entries remain bounded by a polynomial in the length of the input. See [12] for a more thorough discussion of this.

Apparently, the first published algorithm for Hermite normal form that substantially overcame this problem of large integer entries is due to Frumkin [12]. Frumkin's algorithm has a running time that is bounded by a polynomial of degree 6 in the length of the input and uses an algorithm for the computation of solutions of linear Diophantine equations. Shortly thereafter, Kannan and Bachem [23] published another polynomial-time algorithm for the computation of Hermite and Smith normal forms. The Kannan–Bachem method uses a rearrangement in the order of operations of the classical Hermite algorithm, and in so doing they were able to prove a polynomial time and space bound for the algorithm. Chou and Collins [7] modified the Kannan–Bachem procedure and gave a better space bound.

Frumkin's approach uses modular arithmetic to control the size of the entries that arise. Others that have used modular arithmetic in various forms include Hu [18, App. A]; Schrijver [28, Chap. 4]; Domich [9]; Domich, Kannan, and Trotter [10]; and Iliopolous [20]. In §2.1 below, we present and analyze an algorithm for computing the Hermite normal form that is an extension of the Domich, Domich–Kannan–Trotter, and Iliopolous algorithms to nonsquare matrices, where the determinant is replaced by a suitable modulus. We also give a different method for calculating the modulus. The extension is relatively straightforward, but we present it here for completeness and to illustrate the improvements given later in §3.

In §2.2, we extend the methods of Domich, Kannan, and Trotter to get a method for calculation of Smith normal forms of integer matrices. This was first done for square matrices of full rank by Domich [9], and independently by Schrijver [28, Chap. 4]. A similar construction was given in [18, App. A], but it does not seem to be complete. More recently, Iliopolous [20] has described a very similar algorithm (at least for matrices of full rank).

2.1. An algorithm for computing the Hermite normal form. We shall express our results in terms of a function $B(t)$, which will bound the number of bit operations required to do each of two different operations. The first operation is to carry out the extended Euclidean algorithm on two $\lceil t \rceil$ bit integers. The second operation is the application of the Chinese remainder theorem with moduli consisting of all primes less than t. By Theorems 8.20 and 8.21 of [1], we can take $B(t) \ll M(t)\log t$, where $M(t)$ is a monotonic upper bound for the number of bit operations required to multiply two $\lceil t \rceil$ bit integers. By a result of Schönhage and Strassen [27], $M(t) \ll t \log t \log\log t$, so that

$$B(t) \ll t \log^2 t \log\log t.$$

We also assume that $B(t)/t$ is nondecreasing.

Our first result is the following.

THEOREM 2.1. *There exists a deterministic algorithm that receives as input an $m \times n$ integral matrix A of rank m and a positive integer h that is a multiple of* $\mathrm{Det}(\mathcal{L}(A))$, *and produces as output the Hermite normal form of A. If the entries*

of A are bounded in absolute value by T, then the running time of the algorithm is $O(mnB(\log T) + m^2nB(\log h))$ bit operations.

If only the matrix A is given, but not h, then the following corollary gives a bound for the running time to compute the Hermite normal form of A.

COROLLARY 2.2. *There exists a deterministic algorithm that receives as input an $m \times n$ integral matrix A of rank m, and produces as output the Hermite normal form of A. If the entries of A are bounded in absolute value by T, then the running time of the algorithm is $O(m^2nB(m\log(mT)))$ bit operations.*

The corollary follows immediately from Theorem 2.1 and a special case of the following proposition (the general case will be used in the next section).

PROPOSITION 2.3. *There exists a deterministic algorithm that receives as input an $m \times n$ ($m \leq n$) integral matrix A and produces as output the rank r of A over $\mathbb{Q}$ and a positive integer h such that h is a multiple of the product of the invariant factors of A. If the entries of A are bounded in absolute value by T, then $h \leq r^{r/2}T^r$, and the running time of the algorithm is $O(rmnB(m\log(mT)))$ bit operations.*

We now give the proof of Proposition 2.3. Our goal will be to find a nonsingular submatrix of dimension r, where r is the rank of A over $\mathbb{Q}$. The number h will be the determinant of this submatrix, which by the Hadamard inequality is bounded by $r^{r/2}T^r$. The first step in finding such a submatrix is to calculate a number $z = O(m\log(mT))$ such that

$$m^{m/2}T^m < \prod_{\substack{p\leq z \\ p \text{ prime}}} p < m^mT^{2m},$$

and to calculate all of the primes up to z. Finding z can be done using the estimates of [26] and the calculation of the primes can be done in time $O(z)$ by the method of [25]. The number r is the largest dimension of a nonzero subdeterminant of A. For every prime p, the rank of A modulo p does not exceed r, and it equals r if and only if p does not divide some $r \times r$ subdeterminant. Since every nonzero subdeterminant of A is less in absolute value than $\prod_{p\leq z} p$, at least one of the primes $p \leq z$ will have the property that the rank of A modulo p is equal to r.

We now proceed as follows. For each prime $p \leq z$, we use row reduction modulo p to find the rank of A modulo p, and we keep track of the prime p for which this rank is maximal. We also keep track of the indices of a maximal set of linearly independent columns. After exhausting all of the primes $p \leq z$, we will have a maximal set of linearly independent columns of A over $\mathbb{Q}$, the rank r, and a prime p not dividing some $r \times r$ subdeterminant of the matrix formed by this set of columns. We can then apply column reduction modulo p to find a set of r rows which are linearly independent modulo p. The selected rows and columns from the matrix A will form an $r \times r$ nonsingular (over $\mathbb{Q}$) submatrix. Up to this point the number of bit operations is easily seen to be

$$\ll rmn\sum_{p\leq z} B(\log p) \ll rmnB(c\log z)\frac{z}{\log z}, \tag{1}$$

by the prime number theorem. Since $B(t)/t$ is nondecreasing, this gives at most $O(rmnB(z))$ bit operations.

The final step in this algorithm is to compute h, the absolute value of the determinant of this submatrix. For this we simply compute the determinant modulo p for each prime $p \leq z$, (actually we could use a smaller value for z if $r < m$ but this is

unimportant) and use the Chinese remainder theorem to find the determinant modulo $\prod_{p\le z} p$. (We note that similar use of the Chinese remainder theorem can be found in [4] and [24].) Since the absolute value of the determinant does not exceed $\prod_{p\le z} p$, this gives the correct integer value. The determinant can be computed modulo p in $O(r^3)$ operations modulo p, giving a total of at most $O(r^3 B(z))$ bit operations, as in (1). We have already noted at the beginning of this section that the Chinese remainder theorem takes only $O(B(z))$ bit operations. The following lemma (Lemma 2.4) implies that the product of the invariant factors of A is the gcd of *all* $r \times r$ subdeterminants of A, so that it divides h. This will complete the proof of Proposition 2.3.

LEMMA 2.4. *Let A be an $m\times n$ integral matrix of rank r, and let $S = \operatorname{diag}(s_1, \cdots, s_r, 0, \cdots, 0)$ be the Smith normal form of A. For $1 \le i \le r$, let $\gamma_i(A)$ be the* gcd *of all $i \times i$ subdeterminants of A. Then*

$$s_1 = \gamma_1(A)$$
$$s_i = \gamma_i(A)/\gamma_{i-1}(A), \qquad 2 \le i \le r.$$

The lemma follows immediately from the observations that the γ_i's are invariant under unimodular row and column operations, and that S can be computed from A by such operations (see [15, §7.7]).

We note that in the application of Proposition 2.3 to Corollary 2.2, we are assuming that the matrix has full rank m. Thus we need only do the first search until we find a prime p and m linearly independent columns modulo p. This will then form the necessary submatrix whose determinant we calculate as above. The running time of this procedure will differ from that stated in the proposition only by a constant factor.

We now turn to the proof of Theorem 2.1. The first step is to reduce the entries of A modulo h, and then to convert A into a lower triangular matrix L using unimodular column operations followed by reduction modulo h. This proceeds in the standard order, moving from top to bottom and left to right. In order to introduce a zero in the i, j location (where $i < j$), we use the extended Euclidean algorithm to calculate integers r and t such that $ra_{ii} + ta_{ij} = g$, where $g = \gcd(a_{ii}, a_{ij})$ and $|r|, |t| \le h$. We then replace column i by $r \cdot$ (column i) $+ t \cdot$ (column j), and we replace column j by $-a_{ij}/g \cdot$ (column i) $+ a_{ii}/g \cdot$ (column j). This is equivalent to postmultiplication by the $n \times n$ unimodular matrix constructed by embedding the matrix

$$\begin{bmatrix} r & -a_{ij}/g \\ t & a_{ii}/g \end{bmatrix}$$

into the $n \times n$ identity matrix, and will replace a_{ii} by $\gcd(a_{ii}, a_{ij})$ and a_{ij} by 0. After each such column operation, we reduce the entries in the two columns modulo h.

This clearly produces a lower triangular matrix L. The following lemma shows how to reconstruct the diagonal of the Hermite normal form of A from L.

LEMMA 2.5. *Let A be an integral matrix of size $m \times n$ and rank m, $H = (h_{ij})$ its Hermite normal form, and h a positive integer multiple of* $\operatorname{Det}(\mathcal{L}(A))$. *Let $d_1 = h$, and $d_{i+1} = d_i/h_{ii}$, $i = 1, \cdots, m-1$. If $L = (l_{ij})$ is any lower triangular matrix obtained from A by unimodular column operations followed by reduction modulo h, then $h_{ii} = \gcd(d_i, l_{ii})$.*

The proof of this lemma can be found in [14], but it also follows closely the arguments in [10].

To complete the construction of the Hermite normal form $H = (h_{ij})$, we proceed as follows. First we reconstruct the diagonal of H. Let l_{ij}, $i, j = 1, \cdots, m$ be the entries of L. Set $d_1 = h$. Then for $i = 1, \cdots, m$, use the extended Euclidean algorithm to find integers r_i and t_i such that $r_i d_i + t_i l_{ii} = h_{ii}$ where $h_{ii} = \gcd(d_i, l_{ii})$, and put $d_{i+1} = d_i / h_{ii}$. For $i = 1, \cdots, m$, multiply the ith column of L by t_i and reduce modulo d_i. This produces a matrix whose diagonal is the diagonal of H. It follows from [10, Cor. 2.6] that the columns of the resulting matrix generate $\mathcal{L}(A)$. The final step in computing H is to use unimodular column operations to reduce each of the entries below the diagonal modulo the diagonal entry in its corresponding row. The order of operations is to work from top to bottom and from left to right, and after each column operation we reduce the column entries modulo h.

The running time of this algorithm can be estimated as follows. First, the triangularization modulo h takes at most $O(m^2 n B(\log h))$ bit operations since we have $O(mn)$ entries to zero out, and each requires at most $O(mB(\log h))$ operations. The reconstruction of the diagonal takes at most $O(m^2 B(\log h))$ operations. Finally, the remaining steps to reduce the entries below the diagonal takes at most $O(m^3 B(\log h))$ operations. The total number of operations is thus $O(m^2 n B(\log h))$ bit operations, as claimed.

2.2. An algorithm for computing the Smith normal form. In this section we shall present and analyze an algorithm for computing the Smith normal form of an integer matrix. Without loss of generality, we may assume that our matrix is $m \times n$ with $m \leq n$, since the Smith normal form of A is the same as the transpose of the Smith normal form of the transpose of A.

THEOREM 2.6. *There exists a deterministic algorithm that receives as input an $m \times n$ integral matrix A and a number h that is a positive multiple of the products of the invariant factors of A, and produces as output the Smith normal form of A. If A has rank r and entries bounded in absolute value by T, then the running time of the algorithm is $O(mnB(\log T) + rmnB(\log h)\log h)$ bit operations.*

Once again, we can state a result for the case when the number h and the rank r are not provided as part of the input.

COROLLARY 2.7. *There exists a deterministic algorithm that receives as input an $m \times n$ integral matrix A, and produces as output the Smith normal form of A. If A has entries bounded in absolute value by T, then the running time of the algorithm is $O\left(rmn\{B(r\log(rT))r\log(rT) + B(m\log(mT))\}\right)$ bit operations.*

The proof of this corollary is immediate from the theorem above and Proposition 2.3. Since $r \leq m$, a simpler but less accurate bound on the running time is $O\left(m^3 n B(m\log(mT))\log(mT)\right)$.

For the proof of Theorem 2.6, we begin by describing the algorithm to compute the Smith normal form S of A. As in the Hermite normal form algorithm, there are two parts to the algorithm. In the first part, we reduce to a diagonal matrix, proceeding inductively down the diagonal, and in the second part we reconstruct from this diagonal matrix the Smith normal form. To begin the diagonalization procedure, we perform unimodular row and column operations on A followed by reduction modulo h to construct a matrix of the form

$$B = \left[\begin{array}{c|ccc} b_{11} & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & B^* & \\ 0 & & & \end{array}\right], \tag{2}$$

with the further condition that b_{11} divides every entry of the matrix B^*. To accomplish this, we first set $B = A$, and then perform unimodular column operations and reduction modulo h in order to replace b_{11} by the gcd of the entries in the first row, and to introduce zeros into the rest of the first row. We then perform unimodular *row* operations followed by reduction modulo h so as to replace the new b_{11} by the gcd of the entries in the first column. This may destroy the zeros in the first row, but if so, then it reduces b_{11} by a multiplicative factor of at least 2. We repeat this process until all entries (except b_{11}) of the first row and column are zero. If now there exists an entry b^*_{ij} of B^* for which $b_{11} \nmid b^*_{ij}$, then we add row i back to row 1 and repeat the procedure. Once again this will reduce b_{11} by a multiplicative factor of at least 2. Clearly, b_{11} is replaced by a new entry at most $O(\log h)$ times. Hence after $O(mnB(\log h)\log h)$ bit operations, we arrive at a matrix of the form (2) with b_{11} dividing every entry of B^*. We now apply essentially the same procedure to B^* (note that this will not change the first row or column of B).

This procedure could break down if at some point we encounter a first row with all zero entries. In this case we interchange the first row with some nonzero row below it, if such a row exists. If no such row exists, which will happen after at most r steps, then we terminate the first part of the algorithm. The matrix is now a diagonal matrix $\tilde{S} = \mathrm{diag}(\tilde{s}_1, \cdots, \tilde{s}_m)$ with

$$\tilde{s}_1 | \tilde{s}_2 | \cdots | \tilde{s}_r,$$

and

$$\tilde{s}_{r+1} = \tilde{s}_{r+2} = \cdots = \tilde{s}_m = 0.$$

The final part of the algorithm is to reconstruct the Smith normal form S from $\tilde{S}$. To do this, we set $d_1 = h$, and then for $i = 1, \cdots, r$, we set $s_i = \gcd(d_i, \tilde{s}_i)$, and $d_{i+1} = d_i/s_i$ and $s_i = 0$ for $r + 1 \leq i \leq m$. The matrix $S = \mathrm{diag}(s_1, \cdots, s_m)$ is now the Smith normal form of A.

The running time of the diagonalization part of the algorithm requires at most $O(rmnB(\log h)\log h)$ bit operations. Clearly, time for the reconstruction phase is much less, justifying our claim in the theorem for the running time of the complete algorithm.

It remains to prove the correctness of this algorithm, in particular, the reconstruction procedure. But Lemma 2.4 and the proof of Lemma 2.5 can easily be adapted to show this.

3. Triangularization over rings. From the previous section it is clear that the computation of the Hermite normal form of an $m \times n$ matrix can be accomplished in at most mn applications of the extended Euclidean algorithm, combined with at most $O(m^2n)$ operations on integers. The major portion of the work goes into bounding the size of the integers that are involved. In this section we shall take a more theoretical approach to the problem of computing right unimodularly equivalent triangular forms of matrices, with the goal of reducing the total number of ring operations. This is analogous to the results that relate the complexity of matrix inversion [6] and LU factorization [19] to matrix multiplication over fields.

In §3.1, we will assume our ring is a commutative ring with identity in which every ideal is principal (PIR). Under these assumptions we give a construction and running time bound for computing matrices K and L with $AK = L$, where L is lower triangular and K is unimodular. This particular form, with the unimodular matrix on

the left, has the advantage that we do not need to resort to matrix inversions (a more subtle problem over PIRs than over fields). In §3.2 we add additional assumptions on our ring and show how to get the factorization $A = LK$, using matrix inversions. Of course it is obvious that one can get to this formulation by inverting K and multiplying, but the construction we give here yields a better running time.

3.1. Triangularization over principal ideal rings. We take the following model. Let R be a commutative ring with identity in which every ideal is principal (a PIR), and let $M(m, n, k) = M_R(m, n, k)$ be a bound for the number of ring operations required to multiply an $m \times n$ matrix times an $n \times k$ matrix over R. Here, by ring operation we mean an assignment, or an addition, subtraction, or multiplication (but later in §3.2 we shall include division as a ring operation). Over a general PIR, the Hermite normal form of a matrix need not exist, since there is no notion of size. It is still possible, however, to obtain a (not necessarily unique) triangularization $AK = L$ over the ring R, where L is lower triangular and K is unimodular.

Let A be $m \times n$ and let t be the number of columns of zeros of A which we may assume are the last t columns. We define a *primary triangularization* of A as two matrices L and K (over R) such that $AK = L$, L is lower triangular, and K is unimodular and of the form

$$K = \begin{bmatrix} K_1 & 0 \\ 0 & I_t \end{bmatrix}. \tag{3}$$

Let $T(m, n) = T_R(m, n)$ be the number of ring operations required to compute a primary triangularization for an $m \times n$ matrix A. The major result of this section is to give a bound for $T(m, n)$ in terms of $M(m, n, k)$ and $T(1, 2)$. We have written the time bound in terms of the quantity $T(1, 2)$, since that represents the most primitive operation, other than a primary ring operation, which is required to do this triangularization.

The quantity $T(1, 2)$ can be described simply as the number of ring operations required, given ring elements a and b, to find ring elements x, y, l, k, g such that g is a gcd of a and b, with

$$g = ax + by, \qquad 1 = kx + ly, \qquad 0 = -al + bk, \tag{4}$$

or in other words, to find a unimodular matrix $[\begin{smallmatrix} x & -l \\ y & k \end{smallmatrix}]$ such that

$$[a\ b] \begin{bmatrix} x & -l \\ y & k \end{bmatrix} = [g\ 0].$$

The assumptions of a PIR guarantee that the equations (4) are solvable (see [30, Chap. IV, Thm. 33]), but under these general assumptions it is not clear that there is an algorithm in terms of basic ring operations for finding a solution. (Thus it might be the case that $T(1, 2)$ is infinite.) However, if the ring R is the homomorphic image of a Euclidean ring E, then an algorithm for solving (4) over the ring R would be to take any preimages of a and b in E, solve the equation there, and then project back to R. The most important examples of this situation (for our purposes) are the rings $\mathbb{Z}/h\mathbb{Z}$. We refer the reader to §4, where this is used.

We assume throughout that

$$M(m, m, m) \ll m^{\theta}. \tag{5}$$

From (5) and block decompositions, we may easily deduce the following consequences:

$$M(m,n,n) \ll \begin{cases} mn^{\theta-1} & \text{if } m \geq n, \\ n^2 m^{\theta-2} & \text{if } m < n. \end{cases} \tag{6}$$

The current record on the exponent θ is held by Coppersmith and Winograd [8], who proved that $\theta < 2.376$ holds for all fields. An inspection of their proof reveals that, in fact, it holds for all commutative rings with identity. It is also clear that $\theta \geq 2$.

We can now state the primary result of this section.

THEOREM 3.1. *If R is a* PIR, *then*

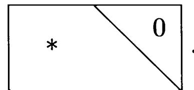
$$T(m,n) \ll \begin{cases} n^2 T(1,2) + mn^{\theta-1} & \textit{if } m \geq n, \\ mnT(1,2) + m^{\theta-1} n \log(2n/m) & \textit{if } m < n. \end{cases}$$

For the sake of comparison, note that the approach of §2.1 implies a bound of the form

$$T(m,n) \ll mnT(1,2) + mn^2, \qquad m \leq n.$$

The proof of Theorem 3.1 uses a block decomposition to get an improvement on the second term. It is perhaps paradoxical that the output data includes a matrix K that is $n \times n$, but that the running time is almost linear in n. A careful examination of the proof reveals that the matrix K that is produced is sparse.

We will prove Theorem 3.1 by reducing the problem to a special case for which we require some notation. Let $\mathcal{S}_m$ be the set of $m \times 2m$ matrices A over R with $a_{ij} = 0$, for $1 \leq i \leq m-1$ and $m+i+1 \leq j \leq 2m$, that is, matrices of the form



Let $T^*(m)$ be the number of ring operations required to produce, for a given $A \in \mathcal{S}_m$, a primary triangularization. We now have the following lemma.

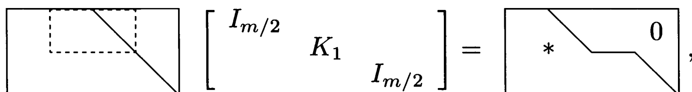
LEMMA 3.2. *We have*

$$T^*(m) \ll m^2 T(1,2) + m^\theta.$$

Proof. By adjoining at most m rows and columns of zeros, we can assume that m is a power of 2 (this is because of the special form of the primary triangularization). The proof proceeds inductively, by reducing the problem to several of half the size. We claim

$$\begin{aligned} T^*(m) &\leq 4T^*(m/2) + 4M(2m,2m,2m) \\ &\leq 4T^*(m/2) + cm^\theta, \end{aligned} \tag{7}$$

for some absolute constant c. The second part of this inequality is immediate from (5). To prove the inequality (7), we begin with a matrix $A \in \mathcal{S}_m$. The first step is to put the top center part (in the dashed box below) of the matrix into triangular form. This transforms the matrix as follows:



where K_1 is $m \times m$ unimodular. After this we triangularize the top left part and the lower right part, giving

$$[\,\cdot\,] \begin{bmatrix} K_2 & \\ & K_3 \end{bmatrix} = \begin{bmatrix} * & 0 \end{bmatrix},$$

where K_2 and K_3 are $m \times m$ unimodular. Finally, we triangularize the bottom center part, which puts the entire matrix into triangular form:

$$[\,\cdot\,] \begin{bmatrix} I_{m/2} & & \\ & K_4 & \\ & & I_{m/2} \end{bmatrix} = \begin{bmatrix} * & 0 \end{bmatrix},$$

where K_4 is unimodular.

To compute the product of the unimodular factors on the left requires at most an extra $4M(2m, 2m, 2m)$ operations. Totalling up all these operations, we get

$$T^*(m) \leq 4T^*(m/2) + 4M(2m, 2m, 2m),$$

which proves the claimed inequality (7).

To complete the proof of the lemma, we iterate (7) to obtain

$$\begin{aligned} T^*(m) &\leq 4T^*(m/2) + cm^\theta \\ &\leq 16T^*(m/4) + cm^\theta + 4c(m/2)^\theta \\ &\ \ \vdots \\ &\leq 4^{\log_2 m} T^*(1) + cm^\theta \sum_{j=0}^{\log_2 m} \left(\frac{4}{2^\theta}\right)^j \\ &\ll m^2 T^*(1) + m^\theta. \end{aligned}$$

But trivially, $T^*(1) = T(1, 2)$, and this proves the lemma. □

We now return to the proof of Theorem 3.1, starting with the case $m = n$. By adjoining m columns of zeros, we have the obvious inequality $T(m, m) \leq T^*(m)$, which by the lemma completes this case. Next, for the case $n < m$, we first put the top $n \times n$ submatrix of A in triangular form. This gives the obvious inequality,

$$T(m, n) \leq T(n, n) + M(m - n, n, n),$$

and by (6) this completes the case $n < m$.

It remains to consider the case $m < n$. By adding at most n columns of zeros, we may assume that $n = 2^k m$ for some positive integer k. Our goal will be to prove the inequality

$$T(m, n) \leq 2T(m, n/2) + P(m, n) + T^*(m) + M(n, 2m, 2m), \tag{8}$$

where $P(m, n)$ denotes the time required to interchange two $m \times n$ blocks in an $n \times n$ matrix, so that clearly $P(m, n) \ll mn$. Before proving (8), we first deduce the theorem from it. From Lemma 3.2 and (6), we obtain

$$\begin{aligned} T(m, n) &\leq 2T(m, n/2) + c_1 m^2 T(1, 2) + c_2 m^{\theta - 1} n \\ &\ \ \vdots \\ &\leq 2^k T(m, m) + \sum_{j=0}^{k} 2^j \left(c_1 m^2 T(1, 2) + c_2 m^{\theta-1} \frac{n}{2^j}\right) \\ &\ll mnT(1, 2) + km^{\theta - 1} n, \end{aligned}$$

which is the conclusion of the theorem.

The proof of (8) should be clear from the following diagram:

$$A = \boxed{\quad * \quad}$$

$$A\begin{bmatrix} K_1 & \\ & K_2 \end{bmatrix} = \boxed{\; * \diagdown \quad 0 \;\Big|\; * \diagdown \quad 0 \;}$$

$$A\begin{bmatrix} K_1 & \\ & K_2 \end{bmatrix} P = \boxed{\; * \diagdown \; 0 \;\Big|\; * \diagdown \quad 0 \;}$$

$$A\begin{bmatrix} K_1 & \\ & K_2 \end{bmatrix} P \begin{bmatrix} K_3 & \\ & I \end{bmatrix} = \boxed{\; * \diagdown \quad 0 \;},$$

where P is an $n \times n$ unimodular matrix which interchanges two $m \times n$ blocks of an $n \times n$ matrix, K_1 and K_2 are $n/2 \times n/2$ unimodular, and K_3 is $2m \times 2m$ unimodular. The term $M(n, 2m, 2m)$ in (8) comes from computing the products of the unimodular factors to the right of A. This completes the proof of (8), and therefore of Theorem 3.1.

3.2. Triangular factorization over principal ideal domains. In this section we impose an additional assumption on our ring and give two results about matrices over the ring. More precisely, we shall assume throughout this section that the ring R is also an integral domain, hence a Principal Ideal Domain (PID). In our operation counts, we include division as a ring operation, with the following stipulation. If $a, b \in R$ and $a \mid b$, then we allow the "operation" b/a to return the $k \in R$ with $b = ka$.

Let $F(m, n) = F_R(m, n)$ be the number of ring operations (including divisions when possible) required to compute for an $m \times n$ matrix A a factorization $A = LK$, where L is lower triangular and K is unimodular of the form (3). Note that $F(1, 2)$ differs very little from $T(1, 2)$, except that the k and l in (4) can be recovered from g, x, and y via $k = a/g$ and $l = b/g$. Finding g, x, and y requires solving the "extended Euclidean problem" $\gcd(a, b) = g = ax + by$ (but of course we do not have a Euclidean algorithm unless R is Euclidean).

We will show that a factorization $A = LK$ is possible with essentially the same running time as in Theorem 3.1. Some result of this type would be obvious if one could reduce the problem of matrix inversion to that of matrix multiplication. In the case when R is a field, this was proved by Bunch and Hopcroft [6]. The following extends that result to integral domains.

THEOREM 3.3. *Let R be an integral domain, and let $I(m) = I_R(m)$ denote the number of ring operations required to compute the inverse of an $m \times m$ invertible matrix A over R. If* (5) *holds, then*

$$I(m) \ll m^{\theta}.$$

The proof of this result over a field given by Bunch and Hopcroft [6] does not appear to extend directly to integral domains. Specifically, their proof uses the fact that if a matrix over a field is invertible, then every row of the matrix contains a unit. This is not the case for an arbitrary ring, as can be seen from the simple example $\left[\begin{smallmatrix} 2 & 3 \\ 3 & 5 \end{smallmatrix}\right]$ over $\mathbb{Z}$.

Proof. In order to prove Theorem 3.3, we shall pass to the field of fractions of R (see [15, p. 51]). Briefly, this field can be defined as follows. Define an equivalence relation $\sim$ on the set of pairs (r, s) of elements of R with $s \neq 0$ by

$$(r, s) \sim (t, u) \quad \text{if and only if } ru = st.$$

The field of fractions then consists of the set of equivalence classes, with addition and multiplication defined by

$$[(r, s)] + [(t, u)] = [(ru + st, su)]$$
$$[(r, s)] \cdot [(t, u)] = [(rt, su)],$$

so that the operations of the field are expressed directly in terms of R.

The first thing we need to show is that the exponent for matrix multiplication θ_F over the field of fractions F is the same as that for matrix multiplication over the ring R, namely, θ in our notation. Let A and B be two $m \times m$ matrices over F and let a and b be two ring elements such that aA and bB are matrices over R (for example, take a to be the product of the denominators of all entries in A). Then the product AB can be computed by first computing $(aA)(bB)$ as matrices over R and then dividing every entry by ab. This requires at most $O(m^2)$ multiplications and divisions to compute aA and bB and AB from $(aA)(bB)$. This is of lower order than the computation of $(aA)(bB)$ over R and shows that $\theta_F \leq \theta$. Conversely, let A and B be matrices over R. Interpret these matrices as defined over F by taking each entry x and replacing it by $[(x, 1)]$ in F. Then the product AB can be computed in F. The final step is to reinterpret the entries of the product as elements of R, by dividing each numerator by its corresponding denominator. This requires an extra $O(m^2)$ operations (actually divisions) and proves the inequality $\theta \leq \theta_F$.

Let F be the field of fractions of R, and let A be an invertible $m \times m$ matrix over R. There is a natural way of viewing the matrices A and A^{-1} as matrices over F, and according to the result of Bunch and Hopcroft, we can compute A^{-1} in $O(m^\theta)$ operations over F, and therefore also in $O(m^\theta)$ operations in R. Once A^{-1} is computed over F, it remains only to interpret the result as a matrix over R, but this can easily be done, since we know that the entries in A^{-1} must be of the form $[(r, s)]$, where $s \mid r$. Hence it suffices to take the elements $[(r, s)]$ and replace them by the corresponding value r/s in R. This completes the proof. □

We now state our result on triangular factorization over PIDs.

THEOREM 3.4. *If R is a* PID, *then*

$$F(m, n) \ll \begin{cases} n^2F(1,2) + mn^{\theta-1} & \text{if } m \geq n, \\ mnF(1,2) + m^{\theta-1}n\log(2n/m) & \text{if } m < n. \end{cases}$$

Note that a straight application of Theorems 3.1 and 3.3 would add an extra factor of n^θ to the above bound.

The proof of this is not much different than that of Theorem 3.1. We first define a quantity $F^*(m)$ to be the number of ring operations required to compute the factorization of a matrix in $\mathcal{S}_m$. Next we claim

$$F^*(m) \leq 4F^*(m/2) + 2I(m) + 4M(2m, 2m, 2m). \tag{9}$$

To see that the claim holds, we need an intermediate observation. Let

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$$

be $m \times m$, with $X_1 \in \mathcal{S}_{m/2}$. If $X_1 = L_1K_1$ is a factorization into lower triangular times unimodular, then

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} L_1K_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} L_1 \\ X_2K_1^{-1} \end{bmatrix} K_1, \tag{10}$$

where now

$$\begin{bmatrix} L_1 \\ X_2K_1^{-1} \end{bmatrix}$$

has the shape

[square shape diagram: upper part with diagonal and 0 above, lower part *]

This procedure requires $F^*(m/2) + I(m) + M(m/2, m, m)$ operations.

With this in hand, the proof of (9) is essentially the same as (7) except that the unimodular factors are on the right sides of the equations.

We deduce from (9) and Theorem 3.3 the simple bound

$$F^*(m) \ll m^2F(1,2) + m^\theta.$$

This and arguments similar to those in the proof of Theorem 3.1 are all that is needed to complete the proof of Theorem 3.4. The only differences are that the unimodular factors are again on the right sides of the equations and the case $n < m$ requires an inversion of an $n \times n$ unimodular matrix.

4. Triangularization over $\mathbb{Z}$. One application of the Hermite normal form is to give a particular triangular basis for the lattice generated by the columns of an integral matrix. In this section we prove a result that produces a triangular basis with nice properties. It is stronger than the result of Theorem 2.1 in one sense, since the running time is smaller, but weaker since it does not produce the unique Hermite normal form. The output still has the property that the entries are moderate in size, so it is useful in some lattice problems. The algorithm ties together methods similar to Corollary 2.2 and the results of §3.1.

In this section we let θ denote an exponent for which (5) holds for $\mathbb{Z}/N\mathbb{Z}$ for every integer $N > 1$. In particular, $\theta < 2.376$ by [8]. Our result is the following.

THEOREM 4.1. *There exists a deterministic algorithm that receives as input an $m \times n$ integral matrix A of rank m, and produces as output a triangular matrix L that is right unimodularly equivalent to A. If A has entries bounded in absolute value by T, then the entries of L satisfy $0 \le l_{ij} < m^{m/2}T^m$, and the running time of the algorithm is $O(m^{\theta-1} n \log(2n/m) B(m \log(mT)))$ bit operations.*

Proof. The algorithm has three stages. In the first stage, we find the determinant h of a nonsingular submatrix of the matrix A in a manner similar to §2.1, but using fast matrix multiplication techniques. In the second stage, we use the algorithm of Theorem 3.1 over $\mathbb{Z}/h\mathbb{Z}$ to find a triangular matrix L' that is right equivalent to A over $\mathbb{Z}/h\mathbb{Z}$. Finally, in the third stage we reconstruct from L' a triangular matrix L that is right equivalent to A over $\mathbb{Z}$.

To begin the first stage of the algorithm, we proceed as in the algorithm of Theorem 2.1 to calculate the number $z = O(m \log(mT))$ and all of the primes $p \le z$. Now we search for a prime $p \le z$ for which the rank of A over $\mathbb{Z}/p\mathbb{Z}$ is m. For this, we use an algorithm of Ibarra, Moran, and Hui [19]. Their algorithm gives (among

other things) the rank of A and a maximal set of linearly independent columns over $\mathbb{Z}/p\mathbb{Z}$. As soon as we find such a prime p, the set of m linearly independent columns over $\mathbb{Z}/p\mathbb{Z}$ provide our $m \times m$ nonsingular submatrix of A when viewed over $\mathbb{Z}$. The running time of their algorithm is $O(m^{\theta-1}n)$ operations in $\mathbb{Z}/p\mathbb{Z}$, for a total of at most

$$\ll m^{\theta-1}n \sum_{p\leq z} B(\log p) \ll m^{\theta-1}nB(z)$$

bit operations.

We now calculate the determinant of this submatrix using the Chinese remainder theorem and the calculation of the determinant modulo all of the primes $p \leq z$. The calculation of the determinants modulo p can again be accomplished with the algorithm of [19], with a total running time of $O(m^{\theta}B(z))$ bit operations, and the Chinese remainder calculation takes $O(B(z))$ bit operations as before.

After this first stage, we have a number $h \leq m^{m/2}T^m$ that is a multiple of $\mathrm{Det}(\mathcal{L}(A))$. In the second stage we now apply the algorithm of Theorem 3.1 over the ring $\mathbb{Z}/h\mathbb{Z}$ (which is a PIR) to find matrices K' and L' over $\mathbb{Z}/h\mathbb{Z}$ such that $AK' = L'$, K' is unimodular, and L' is lower triangular. The only thing we need to describe is how many bit operations are required to do $T(1,2)$. Since this can be done by the extended Euclidean algorithm in $\mathbb{Z}$ on nonnegative integers bounded by h, this is $B(\log h)$. The number of bit operations required for this stage is $O(m^{\theta-1}n\log(2n/m)B(\log h))$.

Finally, in the third stage we proceed as in the proof of Theorem 2.1 to reconstruct from L' a matrix L whose diagonal is the diagonal of the Hermite normal form of A, and whose columns generate $\mathcal{L}(A)$. Note that in this procedure, the entries are reduced modulo some divisor of h (the d_i's) and so will be bounded between 0 and h. The number of operations for this stage is $O(m^2B(\log h))$, since we need m applications of the extended Euclidean algorithm on integers less than or equal to h, combined with $O(m^2)$ operations modulo h. This completes the proof of Theorem 4.1. □

5. A fast probabilistic method for Smith forms over $\mathbb{Z}$, and some open questions. The results of the previous section suggest an interesting problem, namely, whether algorithms for constructing the Smith normal form can be constructed that make use of fast matrix multiplication techniques, achieving a speedup over the methods presented in §2.2. Our results in §4 come close to achieving this for the Hermite normal form, although we are only able to triangularize the matrix. We were unable to find a block decomposition of the Smith normal form that would allow us to incorporate fast matrix multiplication methods, but if one is willing to consider probabilistic methods, then the method of Kaltofen, Krishnamoorthy, and Saunders [21] might be adapted to the problem at hand.

We will briefly describe such a probabilistic algorithm for computing the Smith normal form, based on the method of Kaltofen, Krishnamoorthy, and Saunders. Let A be an $m \times m$ nonsingular integer matrix for which we wish to compute the Smith normal form, and for a matrix B, let B_i denote the upper left $i \times i$ subdeterminant of B. The algorithm uses a parameter k, and can be described as follows.

1. For $i = 1, \cdots, m$, set $g_i = A_i$.
2. For $i = 1, \cdots, k$, do
   (a) choose random $m \times m$ unimodular matrices X and Y.
   (b) calculate $G = XAY$.
   (c) set $g_i = \gcd(G_i, g_i)$, $1 \leq i \leq m-1$.

3. Output the matrix $T = \mathrm{diag}(t_1, \cdots, t_m)$, where $t_1 = g_1$ and $t_i = g_i/g_{i-1}$, $2 \leq i \leq m$.

We should mention that in practice we can take X and Y to be triangular or the product of an upper and a lower triangular matrix. The principle upon which the algorithm is based is the following. In Lemma 2.4, each γ_i is a gcd of all $i \times i$ subdeterminants of the matrix A. We should be able to calculate this gcd by taking the gcd of a few "random" linear combinations of the $i \times i$ subdeterminants, and it turns out that the G_i's are good candidates for such random linear combinations.

The method of Kaltofen, Krishnamoorthy, and Saunders was originally stated for the problem of computing Smith normal forms of matrices over the polynomial ring $F[x]$, where F is a field. In this case they were able to prove that the algorithm has a good probability of producing the correct output, and thus were able to prove that the problem of computing the Smith form over $\mathbb{Q}[x]$ or $GF(p)[x]$ is in RNC^2. The method of proof that they use does not carry over to the integer case, but we suspect that the probability of failure in the above algorithm decreases exponentially with k. If this can be proved, then we could probably achieve an expected running time of $O(m^3 B(m \log(mT)))$ bit operations using standard matrix multiplication. Using fast matrix multiplication methods, we might even be able to lower it to $O(m^\theta B(m \log(mT)))$.

It is perhaps interesting to note that even though we are not able to prove it, the method given above seems to work very well in practice for integer matrices. It does, however, sometimes produce a wrong answer which may be difficult to detect. In some cases, incorrect outputs can be detected immediately because they may not have the correct divisibility property or even be integral. There seems to be no immediate way to *always* detect a wrong answer other than to run some independent algorithm such as that in Theorem 2.6 and compare the answers.

We note that our algorithms for computing the Hermite and Smith normal forms of matrices over $\mathbb{Z}$ do not produce the unimodular multipliers. For square matrices of full rank the multipliers can be recovered with the same asymptotic running time bounds (see, for example, [20]). For matrices that are not square or not of full rank, the multipliers are not unique. The algorithms that are presented here can be modified to produce these multipliers, but at the expense of increased running time. One technique for doing this that was pointed out to us by Lenstra [31], is to permute the columns of A so that $A = [A_1 : A_2]$, where A_1 is $m \times m$ and nonsingular. We then apply the algorithm to the augmented matrix

$$A^* = \begin{bmatrix} A_1 & A_2 \\ 0 & I_{n-m} \end{bmatrix}.$$

This matrix is clearly square and nonsingular, and any unimodular multiplier that triangularizes A^* will also work for A.

Acknowledgment. We would like to thank Don Coppersmith and Robert Guralnick for helpful conversations during the course of this research.

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] W. A. Blankinship, *Algorithm 287, matrix triangulation with integer arithmetic*, Comm. ACM, 9 (1966), p. 513.

[3] ———, *Algorithm 288, solution of simultaneous linear Diophantine equations*, Comm. ACM, 9 (1966), p. 514.

[4] I. Borosh and A. S. Fraenkel, *Exact solutions of linear equations with rational coefficients by congruence techniques*, Math. Comp., 20 (1966), pp. 107–112.

[5] G. H. Bradley, *Algorithms for Hermite and Smith normal form matrices and linear Diophantine equations*, Math. Comp., 25 (1971), pp. 897–907.

[6] J. R. Bunch and J. E. Hopcroft, *Triangular factorization and inversion by fast matrix multiplication*, Math. Comp., 28 (1974), pp. 231–236.

[7] T.-W. J. Chou and G. E. Collins, *Algorithms for the solution of systems of linear Diophantine equations*, SIAM J. Comput., 11 (1982), pp. 687–708.

[8] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Computation, 9 (1990), pp. 251–280. Extended abstract in Proc. 19th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 1–6.

[9] P. D. Domich, *Residual methods for computing Hermite and Smith normal forms*, Ph.D. thesis, Cornell University, Ithaca, NY, 1985.

[10] P. D. Domich, R. Kannan, and L. E. Trotter, *Hermite normal form computation using modulo determinant arithmetic*, Math. Oper. Res., 12 (1987), pp. 50–59.

[11] M. A. Frumkin, *An application of modular arithmetic to the construction of algorithms for solving systems of linear equations*, Soviet Math. Dok., 17 (1976), pp. 1165–1168.

[12] ———, *Polynomial time algorithms in the theory of linear Diophantine equations*, in Fundamentals of Computation Theory, Lecture Notes in Computer Science, 56, Springer-Verlag, New York, 1977, pp. 386–392.

[13] ———, *Complexity questions in number theory*, J. Soviet Math., 29 (1985), pp. 1502–1517.

[14] J. L. Hafner and K. S. McCurley, *A rigorous subexponential algorithm for computation of class groups*, J. Amer. Math. Soc., 2 (1989), pp. 837–850.

[15] B. Hartley and T. O. Hawkes, *Rings, Modules, and Linear Algebra*, Chapman and Hall, London, 1970.

[16] G. Havas and L. S. Sterling, *Integer matrices and abelian groups*, in Lecture Notes in Computer Science, 72, Springer-Verlag, New York, 1979, pp. 431–451.

[17] C. Hermite, *Sur l'introduction des variables continues dans la théorie des nombres*, J. Reine Angew. Math., 41 (1851), pp. 191–216.

[18] T. C. Hu, *Integer Programming and Network Flows*, Addison–Wesley, Reading, MA, 1969.

[19] O. H. Ibarra, S. Moran, and R. Hui, *A generalization of the fast LUP matrix decomposition algorithm and applications*, J. Algorithms, 3 (1982), pp. 45–56.

[20] C. S. Iliopolous, *Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix*, SIAM J. Comput., 18 (1989), pp. 658–669.

[21] E. Kaltofen, M. S. Krishnamoorthy, and B. D. Saunders, *Fast parallel computation of Hermite and Smith forms of polynomial matrices*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 683–690.

[22] M. Kaminski and A. Paz, *Computing the Hermite normal form on an integral matrix*, Tech. Report 417, Department of Computer Science, Technion–Israel Institute of Technology, Haifa, Israel, 1986.

[23] R. Kannan and A. Bachem, *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8 (1979), pp. 499–507.

[24] M. T. McClellan, *The exact solution of systems of linear equations with polynomial coefficients*, J. Assoc. Comput. Mach., 20 (1973), pp. 563–588.

[25] P. Pritchard, *Fast compact prime number sieves (among others)*, J. Algorithms, 4 (1983), pp. 332–344.

[26] J. B. Rosser and L. Schoenfeld, *Sharper bounds for the Chebyshev functions* $\theta(x)$ *and* $\psi(x)$, Math. Comp., 29 (1975), pp. 243–269.

[27] A. Schönhage and V. Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing, 7 (1971), pp. 281–292.

[28] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1985.

[29] H. J. S. Smith, *On systems of linear indeterminate equations and congruences*, Philos. Trans. Roy. Soc. London, 151 (1861), pp. 293–326; in The Collected Papers of Henry John Steven Smith, Volume I, Chelsea, New York, 1965, pp. 367–409.

[30] O. Zariski and P. Samuel, *Commutative Algebra*, Vol. I, Van Nostrand, Princeton, NJ, 1958.

[31] H. W. Lenstra, personal communication, 1990.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1084–1118, December 1991
© 1991 Society for Industrial and Applied Mathematics
007

NONINTERACTIVE ZERO-KNOWLEDGE*

MANUEL BLUM†, ALFREDO DE SANTIS‡, SILVIO MICALI§, AND GIUSEPPE PERSIANO¶

Abstract. This paper investigates the possibility of disposing of interaction between prover and verifier in a zero-knowledge proof if they share beforehand a short random string.

Without any assumption, it is proven that noninteractive zero-knowledge proofs exist for some number-theoretic languages for which no efficient algorithm is known.

If deciding quadratic residuosity (modulo composite integers whose factorization is not known) is computationally hard, it is shown that the NP-complete language of satisfiability also possesses noninteractive zero-knowledge proofs.

Key words. interactive proofs, randomization, zero-knowledge proofs, secure protocols, cryptography, quadratic residuosity

AMS(MOS) subject classifications. 68Q15, 94A60

1. Introduction. *Zero-knowledge proofs.* Recently, Goldwasser, Micali, and Rackoff [GoMiRa] have shown that it is possible to prove that some theorems are true without giving the slightest hint of why this is so. This is rigorously formalized in the somewhat paradoxical notion of a *zero-knowledge proof system* (ZKPS).

Zero-knowledge proofs have proven to be very useful both in Complexity Theory and in Cryptography. For instance, in Complexity Theory, via results of Fortnow [Fo] and Boppana, Hastad, and Zachos [BoHaZa], zero-knowledge provides us an avenue to convince ourselves that certain languages are not NP-complete. In cryptography, zero-knowledge proofs have played a major role in the recently proven completeness theorem for protocols with honest majority [GoMiWi2], [ChCrDa], and [BeGoWi]. They also have inspired rigorously analyzed identification schemes [FeFiSh], [MiSh] that are as efficient as folklore ones.

The ingredients of zero-knowledge. Despite its wide applicability, zero-knowledge remains an intriguing notion: What makes zero-knowledge proofs work?

Three main ingredients differentiate standard zero-knowledge proofs from more traditional ones:

1. *Interaction*: The prover and the verifier talk back and forth.
2. *Hidden Randomization*: The verifier tosses coins that are hidden from the prover and thus unpredictable to him.
3. *Computational Difficulty*: The prover embeds in his proofs the computational difficulty of some other problem.

In sum, quite a rich scenario is needed for implementing zero-knowledge proofs. Can one achieve the same results "with fewer ingredients"? Properly answering this question is the goal of this paper. Any such answer is not only important from a purely

* Received by the editors September 4, 1990; accepted for publication February 15, 1991.

† Computer Science Department, University of California, Berkeley, California 94720. This author's research was supported by National Science Foundation grant # DCR85-13926.

‡ IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York 10598. The work of this author was done at IBM, while he was on leave from Dipartimento di Informatica ed Applicazioni, Università di Salerno, Salerno, Italy.

§ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The works of this author was supported by National Science Foundation grant # CCR-8719689.

¶ Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138. The work of this author was partially supported by Office of Naval Research grant # N00039-88-C-0163.

theoretical point of view, but from a practical one as well: the ability to implement zero-knowledge proofs in "poorer" settings would greatly enhance the applicability of these ideas.

1.1. A new, simpler scenario for zero-knowledge. *The new goal.* Let A and B be two mathematicians. A leaves for a long trip around the world, during which he continues his mathematical investigations. We want to enable him, whenever he discovers the proof of a new theorem, to write a postcard to B proving the validity of his assertion in zero-knowledge. This is a noninteractive process. Better, it is a monodirectional interaction: from A to B only. In fact, even if B would like to answer, he couldn't: A has no stable (or predictable) address and will move away before any mail can reach him.

The new scenario. Achieving the new goal is a bit tricky. Without any shared information, "monodirectional" and zero-knowledge proofs are possible only for trivial statements. We shall see, however, that, under a complexity assumption, such proofs exist for any "NP theorem" thanks to a simple, innocent-looking, ingredient: shared randomness. That is, both prover and verifier have access to the same, short, random string.

Past and present. Blum, Feldman, and Micali [BlFeMi] were the first to conceive that zero-knowledge proofs could be based on the above, simple ingredient, and proposed the name of noninteractive zero-knowledge proofs for them, and presented some noninteractive zero-knowledge proofs. De Santis, Micali, and Persiano [DeMiPe1] improved on their results by using a weaker complexity assumption. The present paper summarizes and improves on both these results.

First, we contribute a crisper formalization of noninteractive zero-knowledge; second, we modify their algorithms and provide a full proof of correctness for them, thus removing a subtle bug (pointed out by Bellare) in some part of their argument.[1]

1.2. Shared random strings and public coins. As we have said, we have prover and verifier share a common, random string. Actually, in our proof systems the verifier will not toss any secret coins at all.

The idea of protocols with public randomness is not new. Protocols making use of public randomness were already known in the literature, both in a cryptographic and in a complexity-theoretic scenario. These protocols, however, were developed for quite different ends, and differ from our scenario in the way the coin tosses are made available.

Random beacons. In [Ra3], Rabin presents the notion of a random beacon. This is a source broadcasting random bits at regular time intervals. He used this device for "achieving simultaneity" in contract signing.

Note, though, that sharing a common random string is a requirement *weaker* than having both parties access a random beacon (e.g., sharing the same Geiger counter). In this latter case, in fact, all made coin tosses would be seen by both parties, but the future ones would still be unpredictable. By contrast, our model allows the prover to see in advance the outcome of all the coin tosses the verifier will ever make. That is, the zero-knowledgeness of our proofs does not depend on the secrecy or unpredictability of σ but on the "well mixedness" of its bits![2]

[1] The part that presented a problem in their argument was the one relative to "many-theorems," that is, the equivalent of our §6.

[2] This curious property makes our result potentially applicable. For instance, all libraries in the country possess identical copies of the random tables prepared by the Rand Corporation. Thus, we may think of ourselves as being already in the scenario needed for noninteractive zero-knowledge

Note that sharing a random string σ is a weaker requirement than being able to interact. In fact, if A and B could interact, they would be able to construct a common random string, for instance, by coin tossing over the phone [Bl1]; the converse, however, is not true.

Arthur–Merlin games. The question of the power of hidden randomness versus public randomness has already been discussed in Complexity Theory in the context of proof systems. Goldwasser, Micali, and Rackoff [GoMiRa] and Babai and Moran [Ba], [BaMo] consider proofs as games played between two players, prover and verifier, who can talk back and forth. In [GoMiRa], the verifier is allowed to flip fair coins and hide their outcomes from the prover. In [Ba], [BaMo], all coin tosses made by the verifier are seen by the prover—called, respectively, Arthur and Merlin in proof systems of this type. Actually, each message from the verifier to the prover consists of a random string. Thus in an Arthur–Merlin proof system, the verifier can be substituted by a random beacon: rather than having the verifier send his next message, one waits for the next transmission of the beacon. That is, once again, all made coin tosses are publicly known, but future ones are still unpredictable. Only if the verifier is guaranteed to send a single message are we in a shared-random-string scenario. The class of languages recognized by such a restricted proof system is denoted by "AM_2" or "$AM[2]$" (to specify that there are exactly two rounds of communication). We show that, under proper complexity assumptions, this class coincides with the set of languages possessing noninteractive zero-knowledge proofs.

1.3. Applications of noninteractive zero-knowledge. Powerful computer networks are in place, and can be used for executing a huge variety of cryptographic protocols. Zero-knowledge proofs are crucial to these protocols and, at the same time, interaction is the most expensive resource.[3] Thus noninteractive zero-knowledge proofs may be used to save precious communication rounds in cryptographic protocols.

Besides this, noninteractive zero-knowledge has been used by Bellare and Goldwasser [BeGo] as an alternative basis for secure digital signatures (in the sense of [GoMiRi]). Also, following a hint of [BlFeMi], Naor and Yung [NaYu] exhibit public-key cryptosystems secure against chosen cipher-text attack.

1.4. Organization. The next section is devoted to setting up our notation, recalling some elementary facts from Number Theory, and stating the complexity assumption which suffices to show the existence of noninteractive ZKPS.

In §3 we define the notion of bounded noninteractive zero-knowledge; that is, the "single theorem" case.

In §4 we show that a special number-theoretic language L possesses a bounded noninteractive zero-knowledge proof. That is, if prover and verifier share a random string, then it is possible to prove, noninteractively and in zero-knowledge, that any single, sufficiently shorter $x \in L$.

In §5, under the quadratic residuosity assumption, we prove that the "more general" language of $3SAT$ is in bounded noninteractive zero-knowledge.

Only in §6 do we show that, if deciding quadratic residuosity is hard, the prover can show in zero-knowledge membership in NP languages for any number of strings, each of arbitrary size, using the *same* randomly chosen string.

In §7 we will discuss some related work.

proofs.

[3] The internal computation of a typical cryptographic protocol can be performed in a few seconds, but the time it takes to exchange electronic mail a hundred times may not be negligible.

In §8 we will state an open problem that we would love to see solved.

2. Preliminaries.

2.1. Basic definitions. *Notation.* We denote by $\mathcal{N}$ the set of natural numbers. If $n \in \mathcal{N}$, by 1^n we denote the concatenation of n 1's. We identify a binary string σ with the integer x whose binary representation (with possible leading zeros) is σ.

By the expression $|x|$ we denote the length of x if x is a string, the length of the binary string representing x if x is an integer, the absolute value of x if x is a real number, or the cardinality of x if x is a set.

If σ and τ are binary strings, we denote their concatenation by either $\sigma \circ \tau$ or $\sigma\tau$.

A language is a subset of $\{0, 1\}^*$. If L is a language and $k > 0$, we set $L_k = \{x \in L\colon |x| \leq k\}$. For variety of discourse, we may call "theorem" a string belonging to the language at hand. (A "false theorem" is a string outside L.)

Models of computation. An algorithm is a Turing machine. An *efficient* algorithm is a probabilistic Turing machine running in expected polynomial time.

We emphasize the number of inputs received by an algorithm as follows. If algorithm A receives only one input, we write "$A(\cdot)$"; if it receives two inputs, we write "$A(\cdot, \cdot)$" and so on.

A sequence of probabilistic Turing machines $\{T_n\}_{n \in \mathcal{N}}$ is an *efficient nonuniform algorithm* if there exists a positive constant c such that, for all sufficiently large n, T_n halts in expected n^c steps and the size of its program is less than or equal to n^c. We use efficient nonuniform algorithms to gain the power of using different Turing machines for different input lengths. For instance, T_n can be used for inputs of length n. The power of nonuniformity lies in the fact that each Turing machine in the sequence may have "wired-in" (i.e., properly encoded in its program) a small amount of special information about its own input length.[4]

A *random selector* is a special (random) oracle. The oracle query consists of a pair of strings $(s, \mathcal{S})$, where the second string encodes a finite set. Such a query is answered by the oracle with a randomly chosen element in the set $\mathcal{S}$. If the oracle is asked the same query twice, it will return the same element. The role of the first entry in the query is to allow us, if so wanted, to make random an independent selection in a set $\mathcal{S}$. That is, if $\mathcal{S}$ is the same, and $s_1 \neq s_2$, then, in response to queries $(s_1, \mathcal{S})$ and $(s_2, \mathcal{S})$, the oracle will return two elements from $\mathcal{S}$, each randomly and independently selected.

A *random selecting algorithm* is a Turing machine with access to a random selector. Note that a random selecting algorithm is strictly more powerful than one with access to coin or random oracle. For instance, a random selecting algorithm can select with uniform probability one out of three elements. On the other hand, simulating independent coin flips is easy with a random selector: If *Select* is a random selector, to ensure the independence of b_i, the ith coin flip, from all the other coin flips in a computation on input x, one can set $b_i = Select(x \circ i, \{0, 1\})$.

Random selectors will simplify the description of our algorithms. In fact, we desire a prover in a noninteractive proof system to be "memoryless." That is, it needs not remember which theorems it proved in the past to find and prove the next theorem. However, for zero-knowledge purposes, it will be much handier to keep track of some history, the history, that is, of previously made coin tosses. This will be crucial in §6. A random selector will, in fact, accomplish this record-keeping without having

---

[4] This definition can be shown to be equivalent to the one of a poly-size combinatorial circuit and to the one [KaLi] of poly-time Turing machine that takes advice.

to consider provers "with history." As we shall point out, random selectors can be efficiently approximated, and thus only represent a conceptual tool.

Algorithms and probability spaces. If $A(\cdot)$ is a probabilistic algorithm, then for any input x, the notation $A(x)$ refers to the probability space that assigns to the string σ the probability that A, on input x, outputs σ.

Following the notation of [GoMiRi], if S is a probability space, then "$x \stackrel{R}{\leftarrow} S$" denotes the algorithm which assigns to x an element randomly selected according to S. If F is a finite set, then the notation "$x \stackrel{R}{\leftarrow} F$" denotes the algorithm which assigns to x an element selected according to the probability space whose sample space is F and uniform probability distribution on the sample points.

If $p(\cdot,\cdot,\cdots)$ is a predicate, the notation $Pr(x \stackrel{R}{\leftarrow} S; y \stackrel{R}{\leftarrow} T; \cdots : p(x,y,\cdots))$ denotes the probability that $p(x,y,\cdots)$ will be true after the ordered execution of the algorithms $x \stackrel{R}{\leftarrow} S$, $y \stackrel{R}{\leftarrow} T, \cdots$.

The notation $\{x \stackrel{R}{\leftarrow} S; y \stackrel{R}{\leftarrow} T; \cdots : (x,y,\cdots)\}$ denotes the probability space over $\{(x,y,\cdots)\}$ generated by the ordered execution of the algorithms $x \stackrel{R}{\leftarrow} S$, $y \stackrel{R}{\leftarrow} T, \cdots$.

2.2. Number theory. *Quadratic Residuosity.* For each integer $x > 0$, the set of integers less than x and relatively prime to x form a group under multiplication modulo x denoted by Z_x^*. We say that $y \in Z_x^*$ is a *quadratic residue* modulo x if and only if there is a $w \in Z_x^*$ such that $w^2 \equiv y \bmod x$. If this is not the case, we call y a *quadratic nonresidue* modulo x. For compactness, we define the *quadratic residuosity predicate* as follows:

$$\mathcal{Q}_x(y) = \begin{cases} 0 & \text{if } y \text{ is a quadratic residue modulo } x \text{, and} \\ 1 & \text{otherwise.} \end{cases}$$

FACT 2.1 (see for instance [NiZu]). If $y_1, y_2 \in Z_x^*$, then

1. $\mathcal{Q}_x(y_1) = \mathcal{Q}_x(y_2) = 0 \Longrightarrow \mathcal{Q}_x(y_1 y_2) = 0$.
2. $\mathcal{Q}_x(y_1) \neq \mathcal{Q}_x(y_2) \Longrightarrow \mathcal{Q}_x(y_1 y_2) = 1$.

The quadratic residuosity predicate defines the following equivalence relation in Z_x^*: $y_1 \sim_x y_2$ if and only if $\mathcal{Q}_x(y_1 y_2) = 0$. Thus, the quadratic residues modulo x form a $\sim_x$ equivalence class. More generally, Fact 2.2 is immediately seen.

FACT 2.2. For any fixed $y \in Z_x^*$, the elements $\{yq \bmod x \mid q$ is a quadratic residue modulo $x\}$ constitute a $\sim_x$ equivalence class that has the same cardinality as the class of quadratic residues.

The problem of deciding quadratic residuosity consists of evaluating the predicate $\mathcal{Q}_x$. As we now see, this is easy when the modulus x is prime and appears to be hard when it is composite.

Prime moduli. Primes are easy to recognize.

FACT 2.3 ([AdHu] extending [GoKi]). There exists an efficient algorithm that, on input x, outputs YES if and only if x is prime.

For p prime, the problem of deciding quadratic residuosity coincides with the problem of computing the Legendre symbol. In fact, for p prime and $y \in Z_p^*$, the Legendre symbol $(y|p)$ of y modulo p is defined as

$$(y|p) = \begin{cases} +1 & \text{if } y \text{ is a quadratic residue modulo } x \text{, and} \\ -1 & \text{otherwise;} \end{cases}$$

and can be computed in polynomial time by using Euler's criterion. Namely,

$$(y|p) = y^{(p-1)/2} \bmod p.$$

Composites are easy to recognize. It is easy to test compositeness.

FACT 2.4 ([Ra1], [SoSt]). There exists a polynomial-time algorithm $TEST(\cdot,\cdot)$ such that

1. if x is composite, $TEST(x,r)$ =COMPOSITE for at least $\frac{3}{8}$ of the strings r such that $|r|=|x|$.
2. if x is prime, $TEST(x,r)$ =PRIME for all r's.

We say that the sequence $(p_1,h_1),\cdots,(p_n,h_n)$ is the *factorization* of x if the p_i's are distinct primes, the h_i's are positive integers, and $x=\prod_{i=1}^{n} p_i^{h_i}$.

While it is easy to test compositeness, no efficient algorithm is known for computing the factorization of a composite integer. In fact, the following assumption is consistent with our state of knowledge.

Factoring assumption. For each efficient nonuniform algorithm $C=\{C_n\}_{n\in\mathcal{N}}$, let p_n^C denote the probability that, on inputing an integer x product of two randomly selected primes of length n, C_n outputs—in some standard encoding—the factorization of x. (This probability is computed over all possible choices of the two primes and the internal coin tosses of C_n.) Then for all positive constants d, and all sufficiently large n, $p_n^C < n^{-d}$.

Often, computational problems relative to composite moduli are easy if their factorization is known. For example, this is the case for the problem of computing square roots modulo x.

FACT 2.5 (see for instance [An]). There exists an efficient algorithm that, given as inputs x, its prime factorization, and y, a quadratic residue modulo x, outputs a random square root of y modulo x.

FACT 2.6 ([Ra2]). The problem of factoring composite integers is probabilistic polynomial-time reducible to the problem of extracting square roots modulo composite integers.

Another computational problem modulo x that is easy given the factorization of x is deciding quadratic residuosity.

FACT 2.7 (see, for instance, [NiZu]). y is a quadratic residue modulo x if and only if y is a quadratic residue modulo each of the prime divisors of x.

However, no efficient algorithm is known for deciding quadratic residuosity modulo composite numbers whose factorization is not given. Some help is provided by the Jacobi symbol, which extends the Legendre symbol to composite integers as follows. Let $(p_1,h_1),\cdots,(p_n,h_n)$ be the prime factorization of x, and $y\in Z_x^*$. Then[5]

$$(y|x)=\prod_{i=1}^{n}(y|p_i)^{h_i}.$$

Define J_x^{+1} and J_x^{-1} to be, respectively, the subsets of Z_x^* whose Jacobi symbol is $+1$ and -1. It can be immediately seen that if $y\in J_x^{-1}$, then it is not a quadratic residue modulo x, as it is not a quadratic residue modulo some prime p_i dividing x. However, if $y\in J_x^{+1}$, no efficient algorithm is known to compute $\mathcal{Q}_x(y)$. Actually, the fastest way known consists of first factoring x and then computing $\mathcal{Q}_x(y)$. This fact was first used in cryptography by Goldwasser and Micali [GoMi1]. We will use it in this paper with respect to the following special moduli.

[5] Despite the fact that the Jacobi symbol is defined in terms of the factorization of the modulus, it can be computed in polynomial time. (This can be derived by a time analysis of the classical algorithm presented in [NiZu]; see also [An].)

Blum integers. For $n \in \mathcal{N}$, we define the set of Blum integers of size n, $BL(n)$, as follows: $x \in BL(n)$ if and only if $x = pq$, where p and q are primes of length n, both congruent to $3 \bmod 4$. These integers were first used for cryptographic purposes by [Bl1].

Blum integers are easy to generate. By Fact 2.3 and the density of the primes congruent to $3 \bmod 4$ (de la Vallee Poussin's extension of the prime number theorem [Sh]), it is easy to prove the following.

FACT 2.8. There exists an efficient algorithm that, on input 1^n, outputs the factorization of a randomly selected $x \in BL(n)$.

This class of integers constitutes the hardest input for any known efficient factoring algorithm. Thus no efficient algorithm is known for deciding quadratic residuosity modulo Blum integers, which justifies the following.

Quadratic Residuosity Assumption (QRA). For each efficient nonuniform algorithm $\{C_n\}_{n \in \mathcal{N}}$, all positive constants d, and all sufficiently large n,

$$Pr\Big(x \stackrel{R}{\leftarrow} BL(n);\ y \stackrel{R}{\leftarrow} J_x^{+1} : C_n(x,y) = \mathcal{Q}_x(y)\Big) < \frac{1}{2} + n^{-d}.$$

That is, no efficient nonuniform algorithm can guess the value of the quadratic residuosity predicate substantially better than by random guessing.

It follows from Fact 2.7 and Euler's criterion that, if x is a Blum integer, $-1 \bmod x$ is a quadratic nonresidue with Jacobi symbol $+1$.

FACT 2.9. On input of a Blum integer x, it is easy to generate a random quadratic nonresidue in J_x^{+1}: randomly select $r \in Z_x^*$ and output $-r^2 \bmod x$.

Regular integers. A Blum integer enjoys an elegant structural property. Namely, $|J_x^{+1}| = |J_x^{-1}|$. More generally, we define an integer x to be *regular* if it enjoys the above property. We define *Regular*(s) to be the set of regular integers with s distinct prime divisors. By the definition of Jacobi symbol, Fact 2.10 is straightforward.

FACT 2.10. An odd integer x belongs to *Regular*(s) if and only if it has s distinct prime factors and is not a perfect square.

Equivalently, by Fact 2.2, we have Fact 2.11.

FACT 2.11. An odd integer x belongs to *Regular*(s) if and only if it is regular and Z_x^* is partitioned by $\sim_x$ into 2^s equally numerous equivalence classes. (Equivalently, J_x^{+1} is partitioned by $\sim_x$ into 2^{s-1} equally numerous equivalence classes.)

3. Bounded noninteractive zero-knowledge proofs. A bounded noninteractive zero-knowledge proof system is a special algorithm. Given as input a random string σ and a single, sufficiently shorter theorem T, it outputs a second string that will convince (noninteractively and) in zero-knowledge that T is true for any verifier who has access to the same σ. It is important in this process that a "brand new" random string is employed for each theorem. The word "bounded" refers to the fact that if the same σ is used over and over again for convincing the verifier of the validity of many theorems, the produced noninteractive proofs may no longer be zero-knowledge.

DEFINITION 3.1. Let A_1 and A_2 be Turing machines. We say that (A_1, A_2) is a *sender–receiver* pair if their computation on a *common input* x works as follows. First, algorithm A_1, on input x, outputs a string m_x. Then, algorithm A_2 computes on inputs x and m_x and outputs ACCEPT or REJECT. If (A_1, A_2) is a sender–receiver pair, A_1 is called the sender and A_2 the receiver. The running time of both machines is calculated only in terms of the common input.

Thus m_x can be interpreted as a message sent by A_1 to A_2.

Notation. In our sender–receiver pairs, the output of the sender is described in terms of s "send instructions," where s depends solely on the input length. If "send v" is the ith such instruction, this is shorthand for "output (i, v)." Without explicitly saying it, the receiver always checks that for each $i = 1, \cdots, s$, exactly one pair with first entry i is received. If this is not the case, or if the second component of a pair is not of the right form (i.e., is not of the proper length, is a string rather than a set, etc.), the receiver immediately halts outputting REJECT. Thus if "send v" is the ith instruction of the sender, "check that $v \cdots$" means "check that the second component of the pair whose first entry is $i \cdots$." That is, the receiver parses without ambiguity the sender's output.

DEFINITION 3.2. Let (*Prover, Verifier*) be a sender–receiver pair where $Prover(\cdot, \cdot)$ is random selecting and $Verifier(\cdot, \cdot, \cdot)$ is polynomial time. We say that (*Prover, Verifier*) is a bounded noninteractive zero-knowledge proof system (bounded noninteractive ZKPS) for the language L if there exists a positive constant c such that:

1. *Completeness.* For all $x \in L_n$ and for all sufficiently large n,

$$Pr(\sigma \overset{R}{\leftarrow} \{0,1\}^{n^c}; Proof \overset{R}{\leftarrow} Prover(\sigma, x) \colon Verifier(\sigma, x, Proof) = 1) > \frac{2}{3}.$$

2. *Soundness.* For all $x \notin L_n$, for all Turing machines $Prover'$, and for all sufficiently large n,

$$Pr(\sigma \overset{R}{\leftarrow} \{0,1\}^{n^c}; Proof \overset{R}{\leftarrow} Prover'(\sigma, x) \colon Verifier(\sigma, x, Proof) = 1) < \frac{1}{3}.$$

3. *Zero-Knowledge.* There exists an efficient algorithm S such that for all $x \in L_n$, for all efficient nonuniform (distinguishing) algorithms D, for all $d > 0$, and, all sufficiently large n,

$$\left| Pr(s \overset{R}{\leftarrow} View(n, x) : D_n(s) = 1) - Pr(s \overset{R}{\leftarrow} S(1^n, x) : D_n(s) = 1) \right| < n^{-d},$$

where

$$View(n, x) = \left\{ \sigma \overset{R}{\leftarrow} \{0,1\}^{n^c}; Proof \overset{R}{\leftarrow} Prover(\sigma, x) \colon (x, \sigma, Proof) \right\}.$$

We call *Simulator* the algorithm S.

We define the class of languages *Bounded*-NIZK as follows:

$$Bounded\text{-NIZK} = \{L \colon L \text{ has a bounded noninteractive ZKPS}\}.$$

A sender–receiver pair (*Prover, Verifier*) is a bounded noninteractive proof system for the language L if there exists a positive constant c such that completeness and soundness hold (such a c will be referred to as the *constant* of (*Prover, Verifier*)). We let *bounded noninteractive P* be the class of languages L having a bounded noninteractive proof system.

We call the "common" random string σ, input to both *Prover* and *Verifier*, the *reference string.* (Above, the common input is σ and x.)

Discussion. *Proving and verifying.* As usual, we do not care how difficult it is to prove a true theorem, but we do insist that verifying is always easy. Thus, we have chosen our prover as powerful as possible, though it cannot use its power to find "long" proofs, since the verifier is polynomial time (in the common input).

Arthur–Merlin games. It is immediately seen that the notion of a bounded noninteractive proof system is equivalent to that of a two-move Arthur–Merlin Proof System [Ba], [BaMo]. Thus, letting AM_2 denote the class of languages accepted by a two-move Arthur–Merlin Proof System, we have *Bounded*-NIZK $\subseteq AM_2$. Actually, as we shall prove in §5.5, this containment is an equality under a proper complexity assumption.

Deterministic verification. Note that our verifiers are defined to be deterministic. Thus, if they want to perform some probabilistic computation, they are forced to use part of the reference string. A cheating prover may thus try to exploit this fact to his advantage.

Probability enhancement. As for the case of BPP algorithms and interactive proofs, the definition of completeness and soundness is independent of the constants $\frac{2}{3}$ and $\frac{1}{3}$. In fact, these (or other "bounded away") probabilities can be pumped up (and down) easily by repeating the proving process sufficiently many times, each using a distinct segment of a sufficiently longer reference string. This process is called "parallel composition." However, as noted by Micali, for the case of interactive zero-knowledge proofs, parallel composition may also enhance the amount of knowledge released! Indeed, zero-knowledge proofs do not appear to be closed under parallel composition. The reason for which straightforward parallel composition fails in the case of interactive zero-knowledge proofs is precisely that the *interaction* may be exploited in subtle ways by a "cheating verifier."[6] One advantage of noninteractive zero-knowledge is exactly the fact that one does not have to worry about "cheating" verifiers: as is immediately seen, bounded noninteractive zero-knowledge is closed under parallel composition.

Completeness means that (after a sufficient enhancement) the probability of succeeding in proving a true theorem T is overwhelming. This is so even if T is selected after the string σ has been chosen. More precisely, a simple counting argument shows that completeness is equivalent to the following:

1$'$. *Strong completeness.* For all probabilistic algorithms *Choose-in-L*$(\cdot)$ that, on inputting an n^c-bit string, return elements in L_n, and all sufficiently large n,

$$Pr(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; \quad x \stackrel{R}{\leftarrow} \textit{Choose-in-L}(\sigma); \quad \textit{Proof} \stackrel{R}{\leftarrow} \textit{Prover}(\sigma, x) : \textit{Verifier}(\sigma, x, \textit{Proof}) = 1) > 1 - 2^{-n}.$$

That strong completeness holds can be seen by first using parallel composition so as to replace the probability $\frac{2}{3}$ of completeness with $1 - 2^{-2n}$, and then noticing that there are at most 2^n theorems of length n.

Actually, completeness can be replaced by an even simpler property. Namely,

1$''$. *Perfect completeness.* For all $x \in L_n$,

$$Pr(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; \textit{Proof} \stackrel{R}{\leftarrow} \textit{Prover}(\sigma, x) : \textit{Verifier}(\sigma, x, \textit{Proof}) = 1) = 1.$$

In fact, we have Theorem 3.3.

THEOREM 3.3. *Let $L \in$ Bounded-NIZK. Then L has a bounded noninteractive ZKPS with perfect completeness.*

Proof. Furer et al. [FuGoMaSiZa] have proved that any AM_2 language has an interactive proof system with perfect completeness. Now let (P, V) be a bounded

[6] Elaborating on this subtle point is not within the scope of this paper. For an explanation of it (and pointers to related results) see [BeMiOs].

noninteractive ZKPS for L for which completeness holds with overwhelming probability. Then modify P as follows. Whenever the proof generated by P is not accepted by the verifier (something that can be easily computed), as bounded noninteractive $P=AM_2$, the new prover interprets the reference string as an Arthur move, and responds with a Merlin move so as to achieve perfect completeness. This extra step guarantees that the verifier will always be convinced (of a true theorem), and thus perfect completeness holds. It is immediately seen that soundness keeps on holding. Also, zero-knowledge keeps on holding: the extra step may be "dangerous," but it is performed only too rarely.

Soundness means that the probability of succeeding in proving a false theorem T is negligible. This still holds if T is chosen after σ has been selected. More precisely, a simple counting argument shows that soundness is equivalent to

2′. *Strong soundness.* For all probabilistic algorithms *Adversary* outputting pairs $(x, Proof)$, where $x \notin L_n$, and all sufficiently large n,

$$Pr(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; (x, Proof) \stackrel{R}{\leftarrow} Adversary(\sigma): \ Verifier(\sigma, x, Proof) = 1) < 2^{-n}.$$

Zero-knowledge guarantees that the proof gives no knowledge but the validity of the theorem. All the verifier may see in our scenario, σ and *Proof*, can be efficiently computed with essentially the same odds without "knowing how to prove T."

Note that in our scenario, the definition of zero-knowledge is simpler than the one in [GoMiRa]. As there is no interaction between verifier and prover, we do not have to worry about possible cheating by the verifier to obtain a "more interesting view." That is, we can eliminate the quantification "$\forall$ *Verifier′*" from the original definition of [GoMiRa].

Analogously to [GoMiRa], we may define a bounded noninteractive proof system (*Prover, Verifier*) to be *perfect zero-knowledge* if the following more stringent condition holds:

3′. *Perfect zero-knowledge.* There exists an efficient algorithm S such that for all $x \in L_n$ and all sufficiently large n,

$$View(n, x) = S(1^n, x),$$

where

$$View(n, x) = \left\{\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; Proof \stackrel{R}{\leftarrow} Prover(\sigma, x): (\sigma, Proof)\right\}.$$

Thus the notion of perfect ZK is independent of the computing power of "the observer/distinguisher."

While for completeness and soundness it is not important whether the true/false theorem is chosen before or after the reference string, this need not to be the case for zero-knowledge. It is actually important that the prover chooses the true theorem T he wants to prove *independently* of σ. This, in practice, is not a restriction, since σ does not have any special meaning. The sole purpose of σ is to provide a common source of randomness, and thus it can be accessed only *after* the prover has chosen which theorem to prove, in which case the "independence" condition is automatically satisfied. Should the prover want to prove a statement "about" the reference string, there is no guarantee that no knowledge would be revealed, while there is still a guarantee that the statement cannot be false.

4. A bounded noninteractive ZKPS for a special language.

DEFINITION 4.1. Set $\mathcal{QR} = \cup_n \mathcal{QR}(n)$ and $\mathcal{NQR} = \cup_n \mathcal{NQR}(n)$, where

$$\mathcal{QR}(n) = \{(x, y) \mid x \in \mathit{Regular}(2),\ |x| \leq n, \text{ and } \mathcal{Q}_x(y) = 0\}$$

and

$$\mathcal{NQR}(n) = \{(x, y) \mid x \in \mathit{Regular}(2),\ |x| \leq n,\ y \in J_x^{+1}, \text{ and } \mathcal{Q}_x(y) = 1\}.$$

If one restricts the modulo x in the definition of $\mathcal{QR}$ and $\mathcal{NQR}$ to be a Blum integer, then the quadratic residuosity assumption states that it is hard to distinguish the languages $\mathcal{QR}$ and $\mathcal{NQR}$.

For $x \in \mathit{Regular}(2)$, QR_x denotes the set $\{y \mid (x, y) \in \mathcal{QR}\}$ and NQR_x the set $\{y \mid (x, y) \in \mathcal{NQR}\}$.

DEFINITION 4.2. If $(x, y) \in \mathcal{NQR}$ and $z \in J_x^{+1}$, we say that $s \in Z_x^*$ is an (x, y)-*root* of z if $z = s^2 \bmod x$ or $zy = s^2 \bmod x$. (Note that only one of the two cases may apply.) If s is an (x, y)-root of z, we write $s = \sqrt[(x,y)]{z}$.

In this section we prove that $\mathcal{NQR}$ has a bounded noninteractive proof system that is perfect zero-knowledge. The proof system below is based on an earlier protocol of Goldwasser and Micali [GoMi2].

The Sender–Receiver Pair (A,B)

Input to A and B:

- $(x, y) \in \mathcal{NQR}(n)$
- A n^3-bit random string ρ.
  (Set $\rho = \rho_1 \rho_2 \cdots \rho_{n^2}$, where each ρ_i has length n.)

Instructions for A:

- For $i = 1, \cdots, n^2$, if $\rho_i \in J_x^{+1}$, then randomly choose and send $s_i = \sqrt[(x,y)]{\rho_i}$.

Instructions for B:

B.0. If $\rho_i \in J_x^{+1}$ for less than $3n$ of the indices i, then stop and ACCEPT. Else,
B.1. Verify that x is odd and that $y \in J_x^{+1}$. If not, stop and REJECT. Else,
B.2. Verify that x is not a perfect square. If not, stop and REJECT. Else,
B.3. If x is a prime power, stop and REJECT. Else,
B.4. For each $\rho_i \in J_x^{+1}$ verify that $s_i = \sqrt[(x,y)]{\rho_i}$. If not, stop and REJECT. Else ACCEPT.

THEOREM 4.3. *(A, B) is a bounded noninteractive* ZKPS *for $\mathcal{NQR}$.*

Proof. First, (A, B) is a sender–receiver pair. Second, B runs in polynomial time. In fact, the Jacobi symbol can be computed in polynomial time, steps B.2 and B.4 are trivial, and step B.3 can be performed as follows:

B.3.1. Compute the largest integer α for which $x = w^\alpha$ for some $w \in \mathcal{N}$. (Only values $1, \cdots, |x|$ should be tried for α and a binary search can be performed for finding w, if it exists.)
B.3.2. Compute z such that $z^\alpha = x$.
B.3.3. If for all $1 \leq i \leq n^2$, $TEST(z, \rho_i)$ =PRIME, stop and REJECT.

Third, properties 1–3 of a bounded noninteractive ZKPS also hold.

Completeness. We actually prove that strong completeness holds. This implies that the weaker property 1 also holds. If $(x, y) \in \mathcal{NQR}(n)$, then step B.1 is trivially passed. Step B.2 is passed because of Fact 2.10. B.3 is passed with probability greater than $1 - 2^{-n}$. This can be argued as follows. For any fixed $\overline{x} \in \mathit{Regular}(2)$,

the probability that $TEST$ outputs PRIME on a single ρ_i is at most $\frac{5}{8}$, and thus (since the ρ_i's are independent) the probability that B.3 is not successfully passed is at most $(\frac{5}{8})^{n^2}$. Since there are at most 2^n x's such that $(x, z) \in \mathcal{NQR}(n)$ for some z, the probability that step B.3 is not successfully passed is at most $2^n(\frac{5}{8})^{n^2} \leq 2^{-n}$. Finally, step B.4 is passed with probability 1. In fact, as $x \in \mathit{Regular}(2)$, by Fact 2.11, there are exactly 2 $\sim_x$ equivalence classes in J_x^{+1}. That is, either ρ_i is a quadratic residue modulo x or ρ_i is in the same equivalence class as y, in which case $y\rho_i$ is a quadratic residue.

Soundness. As for the completeness property, we actually prove that strong soundness holds.

First, observe that B stops at step B.0 only with negligible probability. Indeed, for a fixed $\overline{x}$, the probability that $\rho_i \in J_{\overline{x}}^{+1}$ is greater than $\frac{1}{8}$. By the Chernoff bound (see [AnVa] and [ErSp]), the probability that $\rho_i \in J_{\overline{x}}^{+1}$ for fewer than $3n$ of the indices is (for large n) less than 2^{-2n}. Thus, the probability that there is an x for which B stops at step B.0 is at most $2^n 2^{-2n} = 2^{-n}$.

Assume that $(x, y) \notin \mathcal{NQR}$. Then, either (a) $x \in \mathit{Regular}(2)$ but $\mathcal{Q}_x(y) = 0$, or (b) $x \notin \mathit{Regular}(2)$. For any fixed input $(\overline{x}, \overline{y})$ for which case (a) occurs, the probability that B.4 is successfully passed is at most 2^{-3n}. (In fact, B.4 is passed if and only if all ρ_i's are quadratic residues modulo x.) Thus, the probability that step B.4 is passed, for any input for which case (a) occurs, is at most $2^n 2^{-3n} = 2^{-2n}$.

Consider now the case that $(x, y) \notin \mathcal{NQR}$ because of reason (b). Then either (b.1) x is not regular, or (b.2) $x \in \mathit{Regular}(1)$, or (b.3) $x \in \mathit{Regular}(s)$ for $s \geq 3$. In case (b.1), due to Fact 2.10, an odd x must be a perfect square which would be detected in step B.2. In case (b.2), x is a prime power which would be detected by step B.3. Let us now argue case (b.3). For any fixed $(\overline{x}, \overline{y})$ with $\overline{x} \in \mathit{Regular}(s)$, $s \geq 3$, the probability that step B.4 is successfully passed is at most 2^{-n}. (In fact, this would happen only if, for each $\rho_i \in J_{\overline{x}}^{+1}$, either ρ_i or $\rho_i y$ is a quadratic residue modulo $\overline{x}$. This happens with probability smaller than or equal to $\frac{1}{2}$ since, because of Fact 2.11, there are at least four $\sim_x$ equivalence classes in $J_{\overline{x}}^{+1}$.) Thus the probability that, for any input outside $\mathcal{NQR}$ because of reason (b.3), step B.4 is successfully passed is at most $2^{2n} 2^{-3n} = 2^{-n}$.

Zero-knowledge. Let us specify a (simulating) efficient algorithm M that, on input $(x, y) \in \mathcal{NQR}$, generates a random variable which no algorithm can distinguish from B's view on input $(x, y) \in \mathcal{NQR}$.

M's program
Input: $(x, y) \in \mathcal{NQR}(n)$.
1. Set $Proof$ = empty string.
2. For $i = 1$ to n^2
   Randomly select an n-bit integer s_i, with possible leading 0's.
   If $s_i \notin J_x^{+1}$ then set $\rho_i = s_i$.
      else
         Toss a fair coin.
         If HEAD set $\rho_i = s_i^2 \bmod x$ and append s_i to $Proof$.
         If TAIL set $\rho_i = y^{-1} s_i^2 \bmod x$ and append s_i to $Proof$.
3. Set $\rho = \rho_1 \cdots \rho_{n^2}$.

Output: $(\rho, Proof)$.

Now, let us prove that M is a good simulator for the view of B when interacting with prover A on input $(x, y) \in \mathcal{NQR}$. Actually, (A, B) is *perfect* zero-knowledge.

That is, the random variable output by M is the very same random variable seen by B (and thus the two random variables cannot be distinguished by any nonuniform algorithm, efficient or not). In fact, it can be easily seen that ρ is randomly distributed among the n^3-bit long strings. Moreover, if $\rho_i \in J_x^{+1}$, the corresponding s_i is a random (x, y)-root of ρ_i. Thus s_i has the same probability of belonging to M's output as it has of being sent from prover A to verifier B on inputs (x, y) and ρ. □

Note that the proof system (A, B) does not have *perfect* completeness; that is, there is a negligible probability that the prover, following the protocol, may not succeed in proving a true theorem. We can achieve perfect completeness and still retain perfect zero-knowledge at the expense of further complications which are not necessary in our context.

Robustness of the result. The above proof system is zero-knowledge if the reference string ρ is truly random. We may rightly ask what would happen if ρ is not truly randomly selected. Fortunately, we shall see that the poor randomness of ρ may perhaps weaken the zero-knowledgeness of our proof system, but not its completeness and soundness. In fact, all we require from ρ is that it contain a not too low percentage of quadratic residue and nonresidues modulo any integer in *Regular*(2) of a given length. The same remark applies to all proof systems of this paper. This robustness property is important, as we can never be sure of the quality of our natural sources of randomness.

5. A bounded noninteractive ZKPS for $3SAT$. In this section we exhibit a bounded noninteractive ZKPS for $3SAT$. A boolean formula $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ in conjunctive normal form over the variables $u_1, \cdots, u_k$, where each clause ϕ_i has three literals, is in the language $3SAT$ if it has a satisfying truth assignment $t: \{u_1, \cdots, u_k\} \rightarrow \{0, 1\}$ (see [GaJo] for a more complete treatment). If $\Phi \in 3SAT$, we say that Φ is 3-satisfiable.

The following definition was informally introduced in [BlFeMi], but used in a quite different way.

DEFINITION 5.1. For any positive integer x, define the relation $\approx_x$ on $J_x^{+1} \times J_x^{+1} \times J_x^{+1}$ as follows:

$$(a_1, a_2, a_3) \approx_x (b_1, b_2, b_3) \iff a_i \sim_x b_i \quad \text{for } i = 1, 2, 3.$$

Let $(a_1, a_2, a_3) \approx_x (b_1, b_2, b_3)$. An (a_1, a_2, a_3)-root modulo x (more simply, an (a_1, a_2, a_3)-root, when the modulus x is clear from the context) of (b_1, b_2, b_3) is a triplet (s_1, s_2, s_3) such that $(s_1^2 \bmod x,\ s_2^2 \bmod x,\ s_3^2 \bmod x) = (a_1 b_1 \bmod x,\ a_2 b_2 \bmod x,\ a_3 b_3 \bmod x)$. If $\mathcal{Q}_x(b_1) = \mathcal{Q}_x(b_2) = \mathcal{Q}_x(b_3) = 0$, a square root modulo x (more simply a square root, when the modulus x is clear from the context) of (b_1, b_2, b_3) is a triplet (s_1, s_2, s_3) such that $(s_1^2 \bmod x, s_2^2 \bmod x, s_3^2 \bmod x) = (b_1, b_2, b_3)$.

From Fact 2.11, one can prove the following fact.

FACT 5.2. For each odd integer $x \in$ *Regular*(s), $\approx_x$ is an equivalence relation on $J_x^{+1} \times J_x^{+1} \times J_x^{+1}$ and there are $2^{3(s-1)}$ equally numerous $\approx_x$ equivalence classes.

We write $(a_1, a_2, a_3) \not\approx_x (b_1, b_2, b_3)$ when (a_1, a_2, a_3) is not $\approx_x$ equivalent to (b_1, b_2, b_3).

We now proceed as follows. In §5.1, we describe a sender–receiver pair (P, V). In §§5.2, 5.3, and 5.4 we will prove that (P, V) is a bounded noninteractive ZKPS for $3SAT$.

5.1. The sender–receiver pair (P,V).

Input to P and V:

- A random string $\rho \circ \tau$, where $|\rho| = 8n^3$ and $|\tau| = 2n^4$;
- $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ a 3-satisfiable formula with n clauses over the variables $u_1, u_2, \cdots, u_k$, $k \leq 3n$.

Instructions for P.

P.1. Randomly select $x \in BL(n)$ and $y \in NQR_x$.

P.2. "Prove that $(x, y) \in \mathcal{NQR}(2n)$."
Send the *auxiliary pair* (x, y) and run algorithm A of §4 on inputs (x, y) and ρ. (Call $Proof_1$ the output.)

P.3. "Prove that $\Phi \in 3SAT$."
Let $t\colon \{u_1, \cdots, u_k\} \to \{0, 1\}$ be the lexicographically smallest satisfying assignment for Φ.
Execute procedure `Prove`(Φ, t, x, y, τ) (see below). (Call $Proof_2$ the output.)

<u>**Procedure**</u> `Prove`(Φ, t, x, y, τ)

"$\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ is a 3-satisfiable formula with n clauses over the variables $u_1, u_2, \cdots, u_k$, $k \leq 3n$. $t\colon \{u_1, \cdots, u_k\} \to \{0, 1\}$ is a truth assignment satisfying Φ. $(x, y) \in \mathcal{NQR}(2n)$ and, moreover, $x \in BL(n)$. τ is a $2n^4$-bit long string."

begin{`Prove`}

1. "Break τ into members of J_x^{+1}."
Consider τ as the concatenation of n^3 $2n$-bit integers. If there are fewer than $33n^2$ integers in J_x^{+1} then stop. Else, let $\tau_1, \cdots, \tau_{33n^2}$ be the first $33n^2$ integers belonging to J_x^{+1}.
2. "Assign triplets of elements with Jacobi symbol $+1$ to clauses."
Group the τ_i's in $11n^2$ triplets $(\tau_1, \tau_2, \tau_3), (\tau_4, \tau_5, \tau_6), \cdots$. The first $11n$ triplets are *assigned* to ϕ_1, the second $11n$ triplets are *assigned* to ϕ_2, and so on.
3. "Label the formula Φ."
For each variable u_j, randomly select $r_j \in Z_x^*$ and compute the pairs (u_j, w_j) and $(\overline{u}_j, yw_j \bmod x)$, where

$$w_j = \begin{cases} r_j^2 \bmod x & \text{if } t(u_j) = 0, \text{ and} \\ yr_j^2 \bmod x & \text{if } t(u_j) = 1. \end{cases}$$

We refer to these pairs as the *labeling* of Φ and to w_j ($yw_j \bmod x$) as the *label* of the literal u_j ($\overline{u}_j$).
"Since y is a quadratic nonresidue, by Fact 2.1, yr_j^2 is a quadratic nonresidue. Therefore the label of a literal is a quadratic nonresidue if the literal is true under t."
Send the labeling of Φ.
4. "Prove that Φ is satisfiable."
For each clause ϕ of Φ do:
   - "Randomly select the verifying triplets."
   Let $(\alpha_1, \beta_1, \gamma_1)$ be the labels of the three literals of ϕ.
   Choose at random seven triplets $(\alpha_2, \beta_2, \gamma_2), \cdots, (\alpha_8, \beta_8, \gamma_8)$ in $J_x^{+1} \times J_x^{+1} \times J_x^{+1}$ such that
   (a) $(\alpha_i, \beta_i, \gamma_i) \not\approx_x (\alpha_j, \beta_j, \gamma_j)$ for $1 \leq i < j \leq 8$, and
   (b) $\mathcal{Q}_x(\alpha_2) = \mathcal{Q}_x(\beta_2) = \mathcal{Q}_x(\gamma_2) = 0$.
   Send $(\alpha_1, \beta_1, \gamma_1), \cdots, (\alpha_8, \beta_8, \gamma_8)$.

The triplets $(\alpha_1, \beta_1, \gamma_1), \cdots, (\alpha_8, \beta_8, \gamma_8)$ are the *verifying* triplets of ϕ. "We omit writing $(\alpha_1^\phi, \beta_1^\phi, \gamma_1^\phi), \cdots, (\alpha_8^\phi, \beta_8^\phi, \gamma_8^\phi)$ not to overburden our notation, hoping that clarity is maintained."

- "Prove that $(\alpha_2, \beta_2, \gamma_2)$ is made of quadratic residues."
  Randomly choose and send (s_1, s_2, s_3), a square root of $(\alpha_2, \beta_2, \gamma_2)$.
- For each of the assigned triplets (z_1, z_2, z_3) of ϕ, choose i, $1 \leq i \leq 8$, so that $(z_1, z_2, z_3) \approx_x (\alpha_i, \beta_i, \gamma_i)$. Randomly choose and send a $(\alpha_i, \beta_i, \gamma_i)$-root of (z_1, z_2, z_3).

end{`Prove`}

Instructions for V.

"V receives from P the auxiliary pair (x, y) and two strings $Proof_1$ and $Proof_2$."

V.0. Compute n from $\rho \circ \tau$ and verify that Φ has at most n clauses and each of them has three literals. If not, stop and REJECT. Else,

V.1. Run algorithm B of §4 on inputs ρ, (x, y), and $Proof_1$.
If B stops and rejects, stop and REJECT. Else,

V.2. If `Check_Prove`$(\Phi, x, y, \tau, Proof_2)$=ACCEPT then ACCEPT, else REJECT.

<u>**Procedure**</u> `Check_Prove`$(\Phi, x, y, \tau, Proof_2)$

"$\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ is a formula with n clauses over the variables $u_1, u_2, \cdots, u_k$. x, y are $2n$-bit integers. τ is a $2n^4$-bit long string. $Proof_2$ is a string."

begin{`Check_Prove`}

1. "Verify that the assigned triplets are proper."
   Consider τ as the concatenation of n^3 $2n$-bit integers. If there are fewer than $33n^2$ integers in J_x^{+1} stop and ACCEPT. Else, let $\tau_1, \cdots, \tau_{33n^2}$ be the first $33n^2$ integers belonging to J_x^{+1}.
   "This happens with very low probability."
   Group the τ_i's in $11n^2$ triplets $(\tau_1, \tau_2, \tau_3), (\tau_4, \tau_5, \tau_6), \cdots$. The first $11n$ triplets are *assigned* to ϕ_1, the second $11n$ triplets are *assigned* to ϕ_2, and so on. Verify that they have been properly computed by P.
2. "Verify that Φ has a proper labeling."
   For each variable u_j, verify that the label of the literal $\overline{u}_j$ is equal to the label of the literal u_j multiplied by y modulo x.
3. For each clause ϕ of Φ do:
   3.1. Let $(\alpha_i, \beta_i, \gamma_i)$, $i = 1, \cdots, 8$, be the verifying triplets of ϕ sent by P.
   3.2. Verify that $(\alpha_1, \beta_1, \gamma_1)$ is formed by the labels of the three literals of ϕ.
   3.3. Verify that (s_1, s_2, s_3) is a square root of $(\alpha_2, \beta_2, \gamma_2)$.
   3.4. Verify that for each assigned triplet (z_1, z_2, z_3) of ϕ, you received a $(\alpha_i, \beta_i, \gamma_i)$-root of (z_1, z_2, z_3), for some i, $1 \leq i \leq 8$.
4. If all the above verifications have been successfully made, return ACCEPT; otherwise return REJECT.

end{`Check_Prove`}

5.2. (P,V) is a bounded noninteractive proof system for $3SAT$. First, note that (P, V) is a sender–receiver pair. Further, all checks of V can be performed in polynomial time, since only simple algebraic computations modulo x and a scanning of the strings ρ and τ are needed.

Completeness. The same reasoning done in Theorem 4.3 shows that the probability that V does not REJECT at step V.1 is overwhelming. Let us now consider

step V.2. The verification of the proper labeling of Φ is always passed. Since t is a satisfying truth assignment for Φ, each clause ϕ has at least one literal true under t. This implies that the label of ϕ contains at least one quadratic nonresidue. Because of this, and because there are eight $\approx_x$ equivalence classes, P can compute eight verifying triplets satisfying properties (a) and (b). Moreover, since each $\approx_x$ equivalent class contains a verifying triplet, each assigned triplet is $\approx_x$ equivalent to some $(\alpha_i, \beta_i, \gamma_i)$ and thus possesses an $(\alpha_i, \beta_i, \gamma_i)$-root. Therefore, if check V.1 is passed, so is check V.2.

Soundness. An honest prover chooses the pair (x, y) randomly. A cheating one, though, may choose this pair as function of the reference string. All arguments below thus have the following form. First, we compute the probability that the verifier can be mislead with a fixed pair, and show that this probability is suitably small. Then, we prove that, even summing up over all possible choices of pairs, we still obtain a small probability.

Assume that, in a computation with a cheating prover *Prover'*, V accepts a formula $\Phi \notin 3SAT$. Then, one of the following three events must happen: (a) the pair (x, y) chosen by *Prover'* is not in $\mathcal{NQR}(2n)$; (b) $(x, y) \in \mathcal{NQR}(2n)$, but *Prover'* stops at step P.1 in `Prove`; and (c) $(x, y) \in \mathcal{NQR}(2n)$, *Prover'* does not stop at step P.1 in `Prove`, but Φ is not 3-satisfiable. We shall prove that each of these events is very improbable. The probability that (a) occurs has already been computed in Theorem 4.3 and shown to be exponentially vanishing in n. Now, consider event (b). For each fixed $\overline{x} \in Regular(2), \overline{x} \le n$, since each τ_i has probability greater than or equal to $\frac{1}{8}$ of being in $J_{\overline{x}}^{+1}$, we expect $n^3/8$ such elements in $J_{\overline{x}}^{+1}$. By the Chernoff bound (see [AnVa], [ErSp]), the probability that no more than $33n^2$ belong to $J_{\overline{x}}^{+1}$ is, for large n, at most e^{-n^2}. Thus, the probability that there is an integer x such that case (b) occurs is, for large n, at most $2^{2n}e^{-n^2}$. Let us now consider event (c). If (c) occurs, then the following event (d) must also occur: at least $11n$ consecutive assigned triplets $(\tau_i, \tau_{i+1}, \tau_{i+2})$ must belong to the union of seven $\approx_x$ equivalence classes. In fact, if Φ is not satisfiable, for every labeling of Φ, one of its clauses is labeled with a triplet of quadratic residues (else, all clauses would be satisfiable). Let ϕ be such a clause. Since verification step 3.3 must be passed, *Prover'* must exhibit a square root of $(\alpha_2, \beta_2, \gamma_2)$, and thus this triplet is $\approx_x$ equivalent to ϕ's label, $(\alpha_1, \beta_1, \gamma_1)$. Thus, all verifying triplets of ϕ are contained in the union of at most seven $\approx_x$ equivalence classes. Since each $(\tau_i, \tau_{i+1}, \tau_{i+2})$ is proved in step 3.4 to be $\approx_x$ equivalent to one verifying triplet, then event (d) must be true. The probability of event (d) is at most $7n(0.93)^n$. Indeed, for each fixed $\overline{x}$ the probability that at least $11n$ assigned triplets belong to the union of 7 $\approx_{\overline{x}}$ equivalence classes is less than $7n(\frac{7}{8})^{11n}$; this can be explained as follows: $\frac{7}{8}$ is the probability that each triplet belongs to the union of seven fixed equivalence classes, there are $11n$ triplets, there are at most $\binom{8}{7} = 8$ ways to choose seven classes out of eight, and there are n clauses altogether. Therefore, the probability that there exists an integer x such that case (d) occurs is at most $2^{2n}7n(\frac{7}{8})^{11n} < 7n(0.93)^n$. This concludes the proof of soundness.

Remark. (P, V) can be modified in the same way as (A, B) was to achieve perfect completeness. This is the reason why the verifier in step 1 of `Check_Prove` accepts if there are fewer than $33n^2$ integers in J_x^{+1}. Note also that the prover need not have infinite computing power. In fact, an efficient algorithm can perform all required computations provided that it has as an additional input the satisfying assignment for Φ.

We show now that the proof system (P, V) is also zero-knowledge over $3SAT$.

We first exhibit a simulator for V's view and then prove that it works.

5.3. The simulator. The following algorithm S, on input a formula $\Phi \in 3SAT$ (but not a satisfying assignment for Φ) generates a family of random variables that, under the QRA, no efficient nonuniform algorithm can distinguish from the view of V. Note that the view of V consists of a quadruple $(\rho \circ \tau, (x, y), Proof_1, Proof_2)$; thus, the task of the simulator is to produce a quadruple that cannot be distinguished, under the QRA, from a correct quadruple. Looking ahead, the two crucial points in the strategy of the simulator are:

1. To choose the auxiliary pair (x, y) so that $x \in BL(n)$ but y is a quadratic residue modulo x.
2. To choose a portion of the reference string not at random. Rather, select it from among the strings that do not contain any quadratic nonresidue modulo x in J_x^{+1}.

This strategy is viable because the simulator can choose the reference string (which is instead fixed for the prover) and because it is hard to distinguish between random members of J_x^{+1} and random quadratic residues modulo x.

For a clearer presentation, S's program has been broken down into procedures. To give informal help in reading these procedures, we write z' for a value computed by the simulator, when we want to emphasize that this value is "fundamentally different" from the "corresponding" value z computed by the prover P, though an exponentially long computation may be required to determine this fact.

S's program

Input: a 3-satisfiable formula $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ over the variables $u_1, u_2, \cdots, u_k$, $k \leq 3n$.

1. Randomly select two n-bit primes $p, q \equiv 3 \bmod 4$ and set $x = pq$. Randomly select $r \in Z_x^*$ and set $y' = r^2 \bmod x$. "Call (x, y') the *auxiliary pair.*"
2. Execute procedure `Gen_`ρ`_and_Proof1`(x, y') obtaining the strings ρ' and $Proof_1$.
3. Generate a random $2n^4$-bit string τ.
4. Execute procedure `Gen_Proof2`$(\Phi, x, y', p, q, \tau)$ obtaining the string $Proof_2$.

Output: $(\rho' \circ \tau, (x, y'), Proof_1, Proof_2)$

<u>Procedure</u> `Gen_`ρ`_and_Proof1`(x, y)

"This procedure is used both by the simulator S and, later on, by some probabilistic algorithm. In any call, $x \in BL(n)$ and $y \in J_x^{+1}$. When the procedure is called by the simulator S, y is a quadratic residue modulo x."

begin{`Gen_`ρ`_and_Proof1`}

1. Set $Proof_1$ = empty string.
2. For $i = 1$ to $4n^2$
   Randomly select a $2n$-bit integer s_i, with possible leading 0's.
   If $s_i \notin J_x^{+1}$ then set $\rho_i = s_i$.
   else
   Toss a fair coin.
   If HEAD then set $\rho_i = s_i^2 \bmod x$ and append s_i to $Proof_1$.
   If TAIL then set $\rho_i = y^{-1} s_i^2 \bmod x$ and append $s_i \bmod x$ to $Proof_1$.
3. Set $\rho = \rho_1 \cdots \rho_{4n^2}$.
4. Return$(\rho, Proof_1)$

end{`Gen_`ρ`_and_Proof1`}

Let us now see that sometimes Gen_ρ_and_Proof1 "generates what the legitimate prover would generate."

LEMMA 5.3. *Define* Space1(x, y) *as the probability space generated by the output of* Gen_ρ_and_Proof1 *on input* x, y. *Then, for all* $x \in BL(n)$ *and* $y \in NQR_x$

$$\texttt{Space1}(x,y) = \left\{\rho \xleftarrow{R} \{0,1\}^{8n^3}; Proof_1 \xleftarrow{R} P_Proof1(x,y,\rho) : (\rho, Proof_1)\right\},$$

where P_Proof1 *is* P*'s procedure to compute* $Proof_1$ *(i.e., step* P.2*).*

Proof. Fix $x \in BL(n)$ and $y \in NQR_x$. It can easily be seen that the first component of Gen_ρ_and_Proof1's output is randomly distributed among the $8n^3$-bit long strings. Moreover, if $\rho_i \in J_x^{+1}$, the corresponding s_i is a random (x, y)-root of ρ_i. Thus s_i has the same probability of belonging to Gen_ρ_and_Proof1's output as it has of being sent, at step P.2, from prover P to verifier V on inputs (x, y) and ρ. □

Procedure Gen_Proof2$(\Phi, x, y', p, q, \tau)$

"This procedure is used both by the simulator S and, later on, by some probabilistic algorithm. In any call, $x \in BL(n)$ and $y' \in QR_x$. It returns a string $Proof_2$ that 'proves' that the formula $\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ is 3-satisfiable using the string τ and the pair (x, y') even without knowing any satisfying assignment for Φ."

begin{Gen_Proof2}

0. Set $Proof_2$ = empty string.
1. Consider τ as the concatenation of n^3 $2n$-bit integers. If there are fewer than $33n^2$ integers in J_x^{+1}, stop. Else, let $\tau_1, \cdots, \tau_{33n^2}$ be the first $33n^2$ integers belonging to J_x^{+1}.
   Group the τ_i's in $11n^2$ triplets $(\tau_1, \tau_2, \tau_3), (\tau_4, \tau_5, \tau_6), \cdots$. The first $11n$ triplets are *assigned* to ϕ_1, the second $11n$ triplets are *assigned* to ϕ_2, and so on.
2. For each variable u_j, randomly select $w_j \in NQR_x$ and label the literal u_j with w_j and the literal $\overline{u}_j$ with $y'w_j \bmod x$.
   "Since y' is a quadratic residue, all labels are quadratic nonresidues."
   Append the labeling of Φ to $Proof_2$.
3. For each clause ϕ of Φ do:
   - Let α_1, β_1, and γ_1 be the labels of the three literals of ϕ. Thus, $\alpha_1, \beta_1, \gamma_1 \in NQR_x$.
     Choose at random seven triplets $(\alpha_2, \beta_2, \gamma_2), \cdots, (\alpha_8, \beta_8, \gamma_8)$ in $J_x^{+1} \times J_x^{+1} \times J_x^{+1}$ such that $(\alpha_i, \beta_i, \gamma_i) \not\approx_x (\alpha_j, \beta_j, \gamma_j)$, for $1 \le i < j \le 8$ and $\mathcal{Q}_x(\alpha_2) = \mathcal{Q}_x(\beta_2) = \mathcal{Q}_x(\gamma_2) = 0$.
     Append the triplets $(\alpha_1, \beta_1, \gamma_1), \cdots, (\alpha_8, \beta_8, \gamma_8)$ as the *verifying* triplets of ϕ to $Proof_2$.
   - Randomly choose and append a square root of $(\alpha_2, \beta_2, \gamma_2)$ to $Proof_2$.
   - For each of the assigned triplets (z_1, z_2, z_3) of ϕ, choose i, $1 \le i \le 8$, so that $(z_1, z_2, z_3) \approx_x (\alpha_i, \beta_i, \gamma_i)$. Randomly choose and append an $(\alpha_i, \beta_i, \gamma_i)$-root of (z_1, z_2, z_3) to $Proof_2$.
4. Return$(Proof_2)$

end{Gen_Proof2}

LEMMA 5.4. *Algorithm* S *is efficient.*

Proof. The main body and procedure Gen_ρ_and_Proof1 are computationally trivial. The first two steps of procedure Gen_Proof2 are also quite easy as, due to

Fact 2.9, generating a random quadratic nonresidue in J_x^{+1} is easy when $x \in BL$. Let us now see also that step 3 can always be completed, and efficiently as well. Given that the first verifying triplet has been chosen to be composed by quadratic nonresidues in J_x^{+1} and the second by quadratic residues, it is certainly possible to choose the other six verifying triplets so that all of them belong to eight distinct $\approx_x$ equivalence classes. Moreover, given that the factorization of x is an available input, the remaining part of step 3 can be efficiently executed. □

5.4. (P,V) is zero-knowledge.

THEOREM 5.5. *Under the* QRA, (P, V) *is a bounded noninteractive* ZKPS *for* $3SAT$.

Proof. All that is left to prove is that (P, V) satisfies the zero-knowledge condition. We do this by showing that algorithm S of the previous section simulates the view of the verifier V.

We proceed by contradiction. Assume that there exists a positive constant d, an infinite subset $\mathcal{I} \subseteq \mathcal{N}$, a set $\{\Phi_n\}_{n\in\mathcal{I}}$ such that each Φ_n is a 3-satisfiable formula with n clauses, and an efficient nonuniform "distinguishing" algorithm $\{D_n\}_{n\in\mathcal{I}}$ such that for all $n \in \mathcal{I}$

$$\left|P_S(n) - P_V(n)\right| \geq n^{-d},$$

where

$$P_S(n) = Pr(s \stackrel{R}{\leftarrow} S(1^n, \Phi_n) : D_n(s) = 1)$$

and

$$P_V(n) = Pr(s \stackrel{R}{\leftarrow} View(\Phi_n) : D_n(s) = 1).$$

We derive a contradiction by showing an efficient nonuniform algorithm $\{C_n\}_{n\in\mathcal{I}}$ violating the QRA. On input randomly chosen $x \in BL(n)$ and $y \in J_x^{+1}$, C_n constructs a string $SAMPLE$ which is distributed according to $S(1^n, \Phi_n)$ if $y \in QR_x$, and according to $View(\Phi_n)$ if $y \in NQR_x$. Thus, as the nonuniform algorithm $\{D_n\}_{n\in\mathcal{I}}$ is assumed to distinguish the two probability spaces, this is a violation of QRA.

The Algorithm C_n

"C_n has "wired-in" a formula Φ_n along with t, the lexicographically smaller satisfying truth assignment for Φ_n, a description of D_n, and the probabilities $P_S(n)$ and $P_V(n)$."

Input: (x, y) such that $x \in BL(n)$ and $y \in J_x^{+1}$.

1. Execute procedure `Gen_`ρ`_and_Proof1`(x, y), thus obtaining ρ and $Proof_1$.
2. Execute procedure `Sample_`τ`_and_Proof2`(Φ_n, t, x, y), thus obtaining τ and $Proof_2$.
3. Set $SAMPLE = \big(\rho \circ \tau, (x, y), Proof_1, Proof_2\big)$.
4. If $D_n(SAMPLE) = 1$ then set $b = 1$ else $b = 0$.
5. If $P_S(n) > P_V(n)$ then **Output**(b) else **Output**$(1 - b)$.

Procedure `Sample_`τ`_and_Proof2`(Φ, t, x, y)

"$\Phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ is a 3-satisfiable formula with n clauses over the variables $u_1, u_2, \cdots, u_k$, $k \leq 3n$. $t : \{u_1, u_2, \cdots, u_k\} \to \{0, 1\}$ is a satisfying truth assignment for Φ. $x \in BL(n)$ and $y \in J_x^{+1}$."

begin{`Sample_τ_and_Proof2`}

1. For $i = 1$ to n^3 do:
   randomly select a $2n$-bit integer r_i (with possible leading 0's)
   if $r_i \notin J_x^{+1}$ then set $s_i = r_i$
   else toss a fair coin: if HEAD then set $s_i = r_i^2 \bmod x$; if TAIL then set $s_i = -r_i^2 \bmod x$.
2. Set $Proof_2$= empty string.
3. Let $j_1, \cdots, j_{33n^2}$ be the indices of the first $33n^2$ s_i's belonging to J_x^{+1}.
   If there are fewer than $33n^2$ such integers set $\tau = s_1 \cdots s_{n^3}$ and stop.
   Else, set $\tau_i = s_i$ for all indices i not in $\{j_1, \cdots, j_{33n^2}\}$.
4. Group the j_i's in $11n^2$ triplets $(j_1, j_2, j_3), (j_4, j_5, j_6), \cdots$. Assign the $11n^2$ triplets to the clauses in the following way: the first $11n$ triplets are *assigned* to the first clause, ϕ_1, the second $11n$ triplets are *assigned* to the second clause, ϕ_2, and so on.
5. For each variable u_j, randomly select $v_j \in Z_x^*$ and assign the *label* w_j to the literal u_j and the label $yw_j \bmod x$ to the literal $\overline{u}_j$, where

$$w_j = \begin{cases} -v_j^2 \bmod x & \text{if } t(u_j) = 1, \text{ and} \\ -yv_j^2 \bmod x & \text{if } t(u_j) = 0. \end{cases}$$

   Call Φ' the labeling of Φ. Append Φ' to $Proof_2$.
6. For each clause ϕ of Φ do:
   - Let $-ya^2 \bmod x, -yb^2 \bmod x, -c^2 \bmod x$ be the label of the three literals of ϕ, and a, b, c previously computed values in Z_x^*.
     "We consider only one case, not to overburden our notation. The other cases are treated similarly."
   - Randomly choose 21 elements $a_1, b_1, c_1, \cdots, a_7, b_7, c_7 \in Z_x^*$, and construct the following eight triplets

$$\begin{gathered}(-ya^2 \bmod x, -yb^2 \bmod x, -c^2 \bmod x)\\ (a_1^2 \bmod x, b_1^2 \bmod x, c_1^2 \bmod x)\\ (a_2^2 \bmod x, -b_2^2 \bmod x, c_2^2 \bmod x)\\ (a_3^2 \bmod x, -b_3^2 \bmod x, -c_3^2 \bmod x)\\ (-a_4^2 \bmod x, b_4^2 \bmod x, c_4^2 \bmod x)\\ (-a_5^2 \bmod x, b_5^2 \bmod x, -c_5^2 \bmod x)\\ (-a_6^2 \bmod x, -b_6^2 \bmod x, c_6^2 \bmod x)\\ (ya_7^2 \bmod x, yb_7^2 \bmod x, -c_7^2 \bmod x).\end{gathered}$$

   - Construct the eight *verifying* triplets of ϕ as follows. Set

$$\begin{aligned}(\alpha_1, \beta_1, \gamma_1) &= (-ya^2 \bmod x, -yb^2 \bmod x, -c^2 \bmod x),\\ (\alpha_2, \beta_2, \gamma_2) &= (a_1^2 \bmod x, b_1^2 \bmod x, c_1^2 \bmod x).\end{aligned}$$

     Randomly permute the remaining six triplets and assign them to

$$(\alpha_3, \beta_3, \gamma_3), \cdots, (\alpha_8, \beta_8, \gamma_8).$$

     Append $(\alpha_1, \beta_1, \gamma_1), \cdots, (\alpha_8, \beta_8, \gamma_8)$ to $Proof_2$.
   - Append the triplet (a_1, b_1, c_1) to $Proof_2$ as a square root of $(\alpha_2, \beta_2, \gamma_2)$.
   - For each of the assigned indices (l_1, l_2, l_3) of ϕ,
     Randomly choose one of the eight verifying triplets, say, $(\alpha_k, \beta_k, \gamma_k)$.
     Randomly choose $v_1, v_2, v_3 \in Z_x^*$ and set $\tau_{l_1} = v_1^2\alpha_k \bmod x$, $\tau_{l_2} = v_2^2\beta_k \bmod x$, and $\tau_{l_3} = v_3^2\gamma_k \bmod x$.

Compute and append to $Proof_2$ $(v_1\alpha_k \bmod x, v_2\beta_k \bmod x, v_3\gamma_k \bmod x)$ as an $(\alpha_k, \beta_k, \gamma_k)$-root of $(\tau_{l_1}, \tau_{l_2}, \tau_{l_3})$.

- Set $\tau = \tau_1 \cdots \tau_{n^3}$.

7. Return(τ, $Proof_2$).

end{`Sample_τ_and_Proof2`}

There is no question that $\{C_n\}_{n\in\mathcal{I}}$ is an efficient nonuniform algorithm. Now let `Space2`(Φ_n, t, x, y) be the probability space generated by the output of `Sample_τ_and_Proof2` on input Φ_n, t, x, y. Then, for all $n \in \mathcal{I}$ and for all $x \in BL(n)$, `Space2`(Φ_n, t, x, y) is equal to

$$(*)\begin{cases} \{\tau \xleftarrow{R} \{0,1\}^{2n^4}; Proof_2 \xleftarrow{R} \texttt{Prove}(\Phi_n, t, x, y, \tau) : (\tau, Proof_2)\} \text{ if } y \in NQR_x, \\ \{\tau \xleftarrow{R} \{0,1\}^{2n^4}; Proof_2 \xleftarrow{R} \texttt{Gen_Proof2}(\Phi_n, x, y, p, q, \tau) : (\tau, Proof_2)\} \text{if } y \in QR_x, \end{cases}$$

where p, q are the prime factors of x.

To see $(*)$, note that if $y \in NQR_x$, then the label w_j assigned to each literal u_j by C_n is a random element selected from either NQR_x or QR_x depending on whether $t(u_j)$ is true or false, respectively (this is the same computation performed by `Prove`). If $y \in QR_x$, then the label w_j of literal u_j is always a random element selected from NQR_x (in the same way as `Gen_Proof2` computes it). In both cases the label of $\overline{u}_j$ is $yw_j \bmod x$.

Regardless of the quadratic residuosity of y modulo x, for each clause ϕ of Φ, the eight verifying triplets of ϕ computed by C_n are always selected at random among the triplets of elements in J_x^{+1} that are pairwise not $\approx_x$ equivalent. The first triplet consists of the labels of the three literals of ϕ, and the second triplet is made of three quadratic residues.

The string τ output by C_n is truly random (regardless of the quadratic residuosity of y modulo x). Indeed, each τ_i is randomly selected from the $2n$-bit long strings, and independently of the remaining τ_j's.

Finally, for each clause and each of its assigned triplets $(\tau_{l_1}, \tau_{l_2}, \tau_{l_3})$ the corresponding $(v_1\alpha_k \bmod x,\ v_2\beta_k \bmod x,\ v_3\gamma_k \bmod x)$ is a random $(\alpha_k, \beta_k, \gamma_k)$-root of $(\tau_{l_1}, \tau_{l_2}, \tau_{l_3})$. This completes the proof of $(*)$.

Since $SAMPLE = (\rho \circ \tau, (x, y), Proof_1, Proof_2)$, because of $(*)$ and because of Lemma 5.3, for randomly selected $x \in BL(n)$ and $y \in J_x^{+1}$, $SAMPLE$ is distributed as $View(\Phi_n)$ if $y \in NQR_x$ and as $S(1^n, \Phi_n)$ if $y \in QR_x$. Given our assumption about the efficient nonuniform algorithm $\{D_n\}_{n\in\mathcal{I}}$, it is immediately seen that, for all $n \in \mathcal{I}$, $Pr\big(x \xleftarrow{R} BL(n); y \xleftarrow{R} J_x^{+1} : C_n(x, y) = \mathcal{Q}_x(y)\big) \geq 1/2 + 1/(2n^d)$, which contradicts the QRA. □

Remark. The reader is encouraged to verify that if the same reference string σ and the same (x, y) are used by the prover to prove that two formulae Φ and $\hat{\Phi}$ are 3-satisfiable, then "extra knowledge may leak," for instance, that there exist a satisfying assignment for Φ and a satisfying assignment for $\hat{\Phi}$ for which the literal u_1 in Φ and the literal $\hat{u}_2$ in $\hat{\Phi}$ have the same truth value.

The moral is that one must be careful when using the same set-up, i.e., common reference string, and the same pair (x, y), to prove an "unlimited" number of formulae to be satisfiable. This is indeed the goal of §6.

5.5. Arthur–Merlin games and bounded noninteractive zero knowledge.

THEOREM 5.6. *If $3SAT \in$ Bounded-NIZK, then Bounded-NIZK $= AM_2$.*

Proof. Since *Bounded*-NIZK $\subseteq$ *Bounded noninteractive* $P = AM_2$, it only remains to show that $AM_2 \subseteq$ *Bounded*-NIZK. Let $L \in AM_2$. Then, there exist a positive constant c and a sender–receiver pair (*Prover, Verifier*) such that

1. for all $x \in L_n$,

$$Pr(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; Proof \stackrel{R}{\leftarrow} Prover(\sigma, x):\ Verifier(\sigma, x, Proof) = 1) > \frac{2}{3}$$

and

2. for all $x \notin L_n$, for all Turing machines $Prover'$, and for all sufficiently large n,

$$Pr(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n^c}; Proof \stackrel{R}{\leftarrow} Prover'(\sigma, x):\ Verifier(\sigma, x, Proof) = 1) < \frac{1}{3}.$$

Moreover, by the result of [FuGoMaSiZa], the proof system (*Prover, Verifier*) enjoys perfect completeness. Define now the language $L' = \cup_n L'(n)$, where

$$L'(n) = \{(r, x) : |r| = n^c,\ x \in L_n,\ \text{and } \exists w,\ |w| \le n^c \text{ such that } Verifier(r, x, w) = 1\}$$

and L and c are as above. Then $x \in L_n$ if and only if $(r, x) \in L'(n)$ for most n^c-bit strings r.[7] Moreover, $L' \in NP$, thus there is a fixed polynomial-time computable reduction R such that

$$(r, x) \in L'(n) \iff \Psi = R(r, x) \in 3SAT_{n^b},$$

where $b > 0$ is a fixed constant depending only on the reduction R.

We now describe a bounded noninteractive ZKPS (P, V) for L. On input $x \in L_n$ and the reference string $\tau = r \circ \sigma$, where $|r| = n^c$ and σ has the proper length, P constructs the formula $\Psi = R(r, x)$ and, if it is 3-satisfiable, then runs the algorithm for the prover P of §5.1 with input Ψ and σ, to prove that, indeed, $\Psi \in 3SAT_{n^b}$. □

THEOREM 5.7. *Under the* QRA, *Bounded*-NIZK $= AM_2$.

6. Noninteractive zero-knowledge. We now want to capture the ability of giving noninteractive and zero-knowledge proofs of "many" theorems, using the same common reference string, in an "on-line manner." That is, each theorem can be proven independently of all previous and future theorems.

We will present our formal definition when the theorems to be proven are statements about 3-satisfiability.

DEFINITION 6.1. Let (*Prover, Verifier*) be a sender–receiver pair, where $Prover(\cdot,\cdot)$ is random selecting and $Verifier(\cdot,\cdot,\cdot)$ is polynomial time. We say that (*Prover, Verifier*) is a noninteractive zero-knowledge proof system (noninteractive ZKPS) if the following three conditions hold.

1. *Completeness.* For all $\Phi \in 3SAT$ and all n,

$$Pr\Big(\sigma \stackrel{R}{\leftarrow} \{0,1\}^{n}; Proof \stackrel{R}{\leftarrow} Prover(\sigma, \Phi):\ Verifier(\sigma, \Phi, Proof) = 1\Big) = 1.$$

---

[7] Thus an alternative way of proving that $x \in L_n$ consists of showing that, for a random string r of the proper length, $(r, x) \in L'(n)$. Note, though, that there may be two different strings x and y in L_n such that $(r, x) \in L'(n)$ for all r, but $(r, y) \notin L'(n)$ for some r's. Thus the fact that for a given string r, $(r, x) \in L'(n)$ constitutes additional information about x than just membership in L_n, and this additional information cannot be hidden by a zero-knowledge proof that $(r, x) \in L'(n)$! This is why we impose the conditions that (*Prover, Verifier*) possess perfect completeness.

2. *Soundness.* There exists a constant $c_1 > 0$ such that, for all probabilistic algorithms *Adversary* outputting pairs $(\Phi', Proof')$, where $\Phi' \notin 3SAT$, for all $d > 0$, and for all $n > c_1$,

$$Pr\Big(\sigma \stackrel{R}{\leftarrow} \{0,1\}^n; (\Phi', Proof') \stackrel{R}{\leftarrow} Adversary(\sigma) : Verifier(\sigma, \Phi', Proof') = 1\Big) < n^{-d}.$$

3. *Zero-knowledge.* There exist constant $c_2 > 0$, an efficient algorithm S such that for all $\Phi_1, \Phi_2, \cdots \in 3SAT$, for all efficient nonuniform algorithms D, for all $d > 0$, and all $n > c_2$,

$$\big|Pr(s \stackrel{R}{\leftarrow} View(n, \Phi_1, \Phi_2, \cdots) : D_n(s) = 1) - Pr(s \stackrel{R}{\leftarrow} S(1^n, \Phi_1, \Phi_2, \cdots) : D_n(s) = 1)\big| < n^{-d}$$

where

$$View(n, \Phi_1, \Phi_2, \cdots) = \Big\{\sigma \stackrel{R}{\leftarrow} \{0,1\}^n; \begin{array}{l} Proof_1 \stackrel{R}{\leftarrow} Prover(\sigma, \Phi_1); \\ Proof_2 \stackrel{R}{\leftarrow} Prover(\sigma, \Phi_2); \\ \quad\vdots \\ \quad : (\sigma, Proof_1, Proof_2, \cdots) \end{array}\Big\}.$$

A sender–receiver pair (*Prover, Verifier*) is a noninteractive proof system for $3SAT$ if completeness and soundness hold.

Discussion. First, note that we have set the probability of acceptance of true theorems to be 1, since $3SAT \in NP$. Note also the generality of our definition as it handles any number of formulae of arbitrary size in completeness, soundness, and zero-knowledge. That is, every true theorem can be proven, no matter how long. Of course, longer theorems will have longer proofs. Since the verifier is polynomial-time in the length of the common input, it will have more time to verify that a longer formula is 3-satisfiable. Every false theorem, no matter how long, has negligible probability of being "successfully proved"; however, though the length of the proof grows with the length of the theorem, "negligible" is defined only as a function of the length of the reference string.[8] Finally, every theorem, no matter how long, possesses a zero-knowledge proof. Of course, a longer theorem will have a longer proof and thus the polynomial-time simulator will have more time to simulate the proofs. The zero-knowledgeness of the simulator's proofs holds only for a nonuniform "observer" bounded by the length of the reference string.[9]

The definition of noninteractive ZKPS might be more general if perfect completeness is relaxed to completeness as in §3. In this case the adversary choosing algorithm *Choose-in-L* should be given σ and access to *Prover*'s random selector.

6.1. The sender–receiver pair (P,V). In this subsection we describe a sender-receiver pair (P, V). P can prove in zero-knowledge the 3-satisfiability of any number of 3-satisfiable formulae with n clauses each. Later, we shall show how to use the same protocol to prove any number of formulae, each of arbitrary size.

Before going into a formal description of the proof system, we give an informal view of the protocol.

[8] Which, de facto, is a security parameter.

[9] In particular, if a theorem and its proof are exponentially long (with respect to the reference string), the distinguishing algorithm can compare the actual "view" and the output of the simulator only for a polynomially long prefix.

An informal look at (P,V).

Observation. A crucial observation that will be (implicitly) proved in this section is the following. If many certified auxiliary pairs (x, y) ($x \in BL$ and $y \in NQR_x$) are available, one can use each (x, y) to prove in zero-knowledge that any *single* formula $\Phi_{(x,y)}$ with n clauses is 3-satisfiable using the *same* random string τ. For what we remarked in §5, the same τ and the same auxiliary pair should not be used to prove the 3-satisfiability of two different formulae.

In the light of the above observation, we want to construct a mechanism to achieve the following two goals:

(1) Associating to each formula Φ an auxiliary pair (x^{Φ}, y^{Φ}), of "bounded" size, so that, with overwhelming probability, different formulae are associated to different pairs.

(2) Certifying (x^{Φ}, y^{Φ}), i.e., proving that $x^{\Phi} \in BL$ and $y^{\Phi} \in NQR_{x^{\Phi}}$.

The first goal could be achieved by using the random selector, but the problem of the certification remains. The current mechanism for certifying in zero-knowledge a single auxiliary pair (x, y) using ρ can be extended to handle "a few" more pairs, but not arbitrarily many.[10] Instead, we use a mechanism of recursive nature to simultaneously achieve (1) and (2).

Let us first describe this recursive mechanism for a prover "with memory." Such a prover can construct and store a binary tree of depth n. The left child of each node will also be denoted as the 0-child, and the right one as the 1-child. Thus each node in the tree is labeled with a binary string of length at most $n+1$. The root is labeled 0, and each other node is labeled with string describing the unique path from the root to it. Thus, for instance, the left child of the root has label 00 and rightmost leaf of the tree has label 01^n. With each node (labeled) i, the prover stores a randomly selected auxiliary pair (x_i, y_i). The prover uses (x_i, y_i) for certifying auxiliary pairs of the children of node i, that is, (x_{i0}, y_{i0}) and (x_{i1}, y_{i1}). The first auxiliary pair (x_0, y_0) is certified using string ρ as in §4. For each i, the two pairs $(x_{0b_1\cdots b_i0}, y_{0b_1\cdots b_i0})$, $(x_{0b_1\cdots b_i1}, y_{0b_1\cdots b_i1})$, are certified together as in §5, using the same string τ_1. That is, consider the language $L = \cup_n L(n)$, where

$$L(n) = \{((u_0, v_0), (u_1, v_1)) : u_0, u_1 \in BL(n),\ v_0 \in NQR_{u_0},\ v_1 \in NQR_{v_1}\}.$$

Then $L \in NP$. Thus, there exists a fixed polynomial-time computable function CR such that

$$((u_0, v_0), (u_1, v_1)) \in L(n) \iff \Psi = CR(u_0, v_0, u_1, v_1) \in 3SAT_{n^e},$$

where e is a fixed constant depending only on the reduction CR. More precisely, let T be a polynomial-time Turing machine such that $x \in L$ if and only if there is a "witness" (string) w such that $|w| \leq |x|^e$ and $T(x, w) = 1$. Then, the formula Ψ is obtained by encoding the computation of T as in Cook's theorem, and then reducing it to a 3-satisfiable formula, as Cook suggested [Co]. A well-known property of this reduction is that to each "witness" w one can associate in polynomial time a satisfying assignment for Ψ. In our case the witness consists of the primes in the factorizations of u_0 and u_1 and their proof of primality. The proof (witness) of the primality of

[10] Recall the way ρ is used. If $\rho_i \in QR_x$, a square root of $\rho_i \bmod x$ is given; if $\rho_i \in NQR_x$ a square root of $y\rho_i \bmod x$ is given. In our simulation, however, all ρ_i will be chosen in QR_x. Thus, if we want to carry on the simulation for many pairs (x_i, y_i) we need to construct a ρ solely consisting of quadratic residues modulo $x_1, x_2, \cdots$, which appears very hard to do when the number of x_i's grows large.

a prime p is probabilistically constructed in a standard way: by running algorithm [AdHu] on input p, flipping coins as needed.

We will thus certify $(x_{0b_1\cdots b_i0}, y_{0b_1\cdots b_i0})$, $(x_{0b_1\cdots b_i1}, y_{0b_1\cdots b_i1})$ by showing that the so constructed

$$\Psi_{0b_1\cdots b_i} = CR((x_{0b_1\cdots b_i0}, y_{0b_1\cdots b_i0}), (x_{0b_1\cdots b_i1}, y_{0b_1\cdots b_i1})) \in 3SAT_{n^e}.$$

For each $\Psi_{0b_1\cdots b_i}$, this is done using the proof system of §5, and the *same* string τ_1, which in fact has length $2n^a$, with $a = 4e$.

What have we gained by this? Essentially, we have transformed the problem of *certifying* $(x_{0b_1\cdots b_i0}, y_{0b_1\cdots b_i0})$, $(x_{0b_1\cdots b_i1}, y_{0b_1\cdots b_i1})$ into the problem of *proving* that $\Psi_{0b_1\cdots b_i} \in 3SAT_{n^e}$, and we have observed (but not yet proved) that one can prove in zero-knowledge arbitrarily many theorems of size n given arbitrarily many independent certified pairs (x, y)'s. Since these pairs are randomly and independently selected, with overwhelming probability, each pair $(x_{0b_1\cdots b_i}, y_{0b_1\cdots b_i})$ is used only once with τ_1 to prove $\Psi_{0b_1\cdots b_i} \in 3SAT_{n^e}$.

In sum, this mechanism provides each formula Φ with a certified auxiliary pair (x^Φ, y^Φ) that is uniquely determined from Φ and the reference string, though still random.

The prover we just described need not remember the labeled full binary tree; it can, in fact, (re)grow its branches as needed. It must, though, remember which auxiliary pairs it had associated with the nodes of the tree. In fact, if it does not keep track of these pairs, it may use the same auxiliary pair and the same reference string to prove different theorems, which may not be zero-knowledge. To avoid this, and to avoid "memory," the prover uses the random selector to associate a random pair with the node of the tree. Namely, on input a formula Ψ the prover chooses n bits $b_1 b_2 \cdots b_n$ by querying the random selector with a pair whose first entry is Ψ and the reference string $\sigma = \rho \circ \tau_1 \circ \tau_2$, and whose second entry is (a description of) the set $\{0,1\}^n$. This way, if the same formula is considered twice, the same random n-bit string would be selected. Then the prover computes a random, first auxiliary pair (x_0, y_0) (again using the random selector so that it could recompute the same pair whenever it wanted to). Then, for $i = 0, \cdots, n$, the auxiliary pairs $(x_{0b_1\cdots b_i0}, y_{0b_1\cdots b_i0})$, $(x_{0b_1\cdots b_i1}, y_{0b_1\cdots b_i1})$, are chosen by the random selector on input $0b_1 \cdots b_i0$ and $0b_1 \cdots b_i1$, respectively. The pair associated with Φ is $(x_{0b_1\cdots b_n}, y_{0b_1\cdots b_n})$.

We now proceed more formally.

Description of (P,V).

"$a = 4e$, where e is the constant of reduction CR. *Select* is P's random selector. $PAIR(n)$ is the set of pairs (x, y) such that $x \in BL(n)$ and $y \in NQR_x$."

Input to P and V:

- A random string σ, $\sigma = \rho \circ \tau_1 \circ \tau_2$, where $|\rho| = 8n^3$, $|\tau_1| = 2n^a$ and $|\tau_2| = 2n^4$.
- A formula $\Phi \in 3SAT$ with n clauses.

Instructions for P:

P.1. "Choose and certify the first auxiliary pair."
Compute auxiliary pair $(x_0, y_0) = Select(\sigma, PAIR(n))$.
Send (x_0, y_0) and run algorithm A of §4 on input (x_0, y_0) and ρ. "Call $Proof_0$ the output."

P.2. "Choose and certify other auxiliary pairs."
Set $b_0 = 0$. Compute and send $b_0 b_1 b_2 \cdots b_n$ =Select$(\Phi, \{0,1\}^n)$.
For $i = 0, \cdots, n$ do:

Set $s = b_0 \cdots b_i$.
Compute and send $(x_{s0}, y_{s0}) = Select(s0, PAIR(n))$ and $(x_{s1}, y_{s1}) = Select(s1, PAIR(n))$.
Compute $\Psi_s = CR(x_{s0}, y_{s0}, x_{s1}, y_{s1})$ and t_s, a satisfying assignment for Ψ_s.
Execute $\texttt{Prove}(\Psi_s, t_s, x_s, y_s, \tau_1)$. "Call $Proof\Psi_s$ the output."

P.3. "Prove $\Phi \in 3SAT$."
Set $s = b_0 \cdots b_n$. Let t_Φ be the lexicographically smaller satisfying assignment for Φ.
Execute $\texttt{Prove}(\Phi, t_\Phi, x_s, y_s, \tau_2)$. "Call $Proof\Phi$ the output."

Instructions for V:

"V receives from P the bits $b_0, b_1, \cdots, b_n$, (x_{b_0}, y_{b_0}), $(x_{b_0 0}, y_{b_0 0})$, $(x_{b_0 1}, y_{b_0 1}), \cdots$, $(x_{b_0 \cdots b_{n-1} 0}, y_{b_0 \cdots b_{n-1} 0})$, $(x_{b_0 \cdots b_{n-1} 1}, y_{b_0 \cdots b_{n-1} 1})$, the formulae $\Psi_{b_0}, \cdots, \Psi_{b_0 b_1 \cdots b_n}$, and the strings $Proof_0$, $Proof\Psi_{b_0}, \cdots, Proof\Psi_{b_0 \cdots b_n}$, $Proof\Phi$."

V.1. "Verify first auxiliary pair."
Run algorithm B of §4 on input ρ, (x_0, y_0), and $Proof_0$.
If B stops and rejects, stop and REJECT. Else,

V.2. "Verify other auxiliary pairs."
For $i = 1, \cdots, n$ do:
Set $s = b_0 \cdots b_i$.
Compute $\Psi_s = CR(x_{s0}, y_{s0}, x_{s1}, y_{s1})$.
If $\texttt{Check_Prove}(\Psi_s, x_s, y_s, \tau_1, Proof\Psi_s)$=REJECT then stop and REJECT. Else,

V.3. "Verify $Proof\Phi$."
Compute n from $\rho \circ \tau_1 \circ \tau_2$ and verify that Φ has at most n clauses, and each of them has three literals. If not, stop and REJECT. Else,
Set $s = b_0 \cdots b_n$.
If $\texttt{Check_Prove}(\Phi, x_s, y_s, \tau_2, Proof\Phi)$=REJECT then stop and REJECT. Else ACCEPT.

6.2. (P,V) is a noninteractive proof system for $3SAT$. The proof system (P, V) of §5 constitutes the main building block of the just-described sender–receiver pair (P, V). Therefore, the completeness of (P, V) can be easily derived from the analysis of completeness in §5.2.

Let us now focus our attention on the soundness. We shall show that, if the formula Φ is not 3-satisfiable, then for any Turing machine *Adversary* (even a "cheating" one that chooses Φ after seeing the reference string), V will accept the proof provided by *Adversary* with sufficiently low probability. The proof closely follows the reasoning done in §5.2 to prove the soundness of the proof system (P, V) described in §5.1. We distinguish two cases:

1. For some w, $(x_w, y_w) \notin \mathcal{NQR}(2n)$.
2. All the pairs (x_w, y_w) belong to $\mathcal{NQR}(2n)$ but $\Phi \notin 3SAT$.

If $(x_0, y_0) \notin \mathcal{NQR}(2n)$, we are in the very same situation analyzed in case (a) in the proof of soundness of §5.2. By the same reasoning, we conclude that the verification of step 1 is passed with sufficiently low probability. Suppose that for $w = sb$, where $b \in \{0, 1\}$, $(x_w, y_w) \notin \mathcal{NQR}(2n)$, and $(x_w, y_w) \in \mathcal{NQR}(2n)$. Then, $\Psi_w \notin 3SAT$ and therefore the procedure $\texttt{Check_Prove}$ invoked for Ψ_w returns REJECT with sufficiently high probability.

Now, suppose that all pairs (x_w, y_w) belong to $\mathcal{NQR}(2n)$ but $\Phi \notin 3SAT$. Since $(x_s, y_s) \in \mathcal{NQR}(2n)$, $s = b_0 b_1 \cdots b_n$, following the reasoning done for cases (b) and (c) in the proof of soundness in §5.2, we conclude that verification step V.3 is passed with very low probability.

Now, we show that the proof system (P, V) is also zero-knowledge over $3SAT$.

6.3. The simulator. In this section, we describe an efficient algorithm S; in the next section we will prove that, on input of a sequence of 3-satisfiable formulae, S's output cannot, under the QRA, be distinguished from V's view by any efficient nonuniform algorithm.

S's Program

Input: An integer $n > 0$. A sequence $\Phi_1, \Phi_2, \cdots$ of 3-satisfiable formulae with n clauses each.

0. Set Sim_Output = empty string and $Tree$ = empty set.
1. "Choose ρ' and choose and certify first auxiliary pair."
   Randomly select two n-bit primes $p_0, q_0 \equiv 3 \bmod 4$ and set $x_0 = p_0 q_0$. Randomly select $y'_0 \in QR_{x_0}$.
   Execute procedure `Gen_ρ_and_Proof1`(x_0, y'_0), thus obtaining the strings ρ' and $Proof'_0$.
2. "Choose τ_1 and τ_2."
   Randomly select two strings τ_1 and τ_2 so that $|\tau_1| = 2n^a$ and $|\tau_2| = 2n^4$.
3. For each input formula Φ do:
   3.1. "Choose and certify other auxiliary pairs"
   Set $b_0 = 0$ and randomly select $b_1 \cdots b_n$. Append (x_0, y'_0), $Proof'_0$, and $b_0 b_1 \cdots b_n$ to Sim_Output. For $i = 0, \cdots, n$ do:
      Let $s = b_0 b_1 \cdots b_i$.
      If $s \notin Tree$ then
         Add s to $Tree$.
         Randomly select four n-bit primes $p_{s0}, q_{s0}, p_{s1}, q_{s1} \equiv 3 \bmod 4$.
         Set $x_{s0} = p_{s0} q_{s0}$ and $x_{s1} = p_{s1} q_{s1}$.
         Randomly select $y'_{s0} \in QR_{x_{s0}}$ and $y'_{s1} \in QR_{x_{s1}}$.
         Compute $\Psi_s = CR(x_{s0}, y'_{s0}, x_{s1}, y'_{s1})$.
         Execute procedure `Gen_Proof2`$(\Psi_s, x_s, y'_s, p_s, q_s, \tau_1)$, thus obtaining $Proof\Psi'_s$.
      Append (x_{s0}, y'_{s0}), (x_{s1}, y'_{s1}), and $Proof\Psi'_s$ to Sim_Output.
   3.2. "Prove $\Phi \in 3SAT$."
   Set $s = b_0 b_1 \cdots b_n$. Execute `Gen_Proof2`$(\Phi, x_s, y'_s, p_s, q_s, \tau_2)$ obtaining $Proof\Phi'$.
   Append $Proof\Phi'$ to Sim_Output.

Output: $(\rho' \circ \tau_1 \circ \tau_2, Sim_Output)$

LEMMA 6.2. *Algorithm S is efficient.*

Proof. The running time of S is proportional to the number of input formulae. For each single input formula, all operations can be efficiently computed. Thus, S is efficient. (Note, again, that the running time is polynomial with respect to the input size, though it may be exponential in the parameter n.) □

The random variable output by S is certainly different from $View$ and, before proceeding any further, let us compare them. In $View$ the string ρ is truly random, while the corresponding string ρ' constructed by S does not contain any element in NQR_{x_0}. In $View$, each y_s is a quadratic nonresidue modulo the corresponding x_s,

whereas in S, y'_s is chosen among the quadratic residues modulo x_s. Because of the different quadratic residuosity of the y_s's, the two distributions differ also in the Ψ_s's and in the strings $Proof\Psi_s$ and $Proof\Phi$. In fact, the formula Ψ_s is satisfiable if and only if both (x_{s0}, y_{s0}) and (x_{s1}, y_{s1}) are of the prescribed form. This is certainly the case in $View$. But in S, as all y_s's are quadratic residues, none of the pairs (x_s, y_s) is of the prescribed form and therefore none of the Ψ_s's is satisfiable. Moreover, the y_s's are also used to compute the labeling of the literals in the strings $Proof\Psi_s$'s and $Proof\Phi$'s and thus in S all literals are labeled with quadratic nonresidues.

In the next section, we shall prove, using a reasoning similar to the one in Section 5.3 that, despite the differences described above, the two families of random variables cannot be distinguished by any efficient nonuniform algorithm, under the QRA.

6.4. (P,V) is zero-knowledge.

THEOREM 6.3. *Under the* QRA, *the sender–receiver pair* (P, V) *of* §6.1 *is a noninteractive ZKPS.*

Proof. All that is left to prove is that (P, V) satisfies the zero-knowledge condition. We do this by showing that the output of algorithm S of the previous section cannot be distinguished from the view of the verifier V by any efficient nonuniform algorithm.

We proceed by contradiction. Assume that there exists a constant $d > 0$, an infinite subset $\mathcal{I} \subseteq \mathcal{N}$, a set $\{(\Phi_1^n, \Phi_2^n, \cdots)\}_{n\in\mathcal{I}}$ of sequences of 3-satisfiable formulae, where Φ_i^n has n clauses, and an efficient nonuniform algorithm $D = \{D_n\}_{n\in\mathcal{I}}$ such that for all $n \in \mathcal{I}$

$$\left|P_V(n) - P_S(n)\right| \geq n^{-d},$$

where

$$P_V(n) = Pr(s \stackrel{R}{\leftarrow} View(\Phi_1^n, \Phi_2^n, \cdots) : D_n(s) = 1)$$

and

$$P_S(n) = Pr(s \stackrel{R}{\leftarrow} S(1^n, \Phi_1^n, \Phi_2^n, \cdots) : D_n(s) = 1).$$

Let $R(n)$ be a polynomial such that the running time and the size of the program of each algorithm D_n is bounded by $R(n)$. Without loss of generality we can consider $R(n)$-tuples of 3-satisfiable formulae $\Phi_1^n, \cdots, \Phi_{R(n)}^n$, instead of arbitrary sequences of 3-satisfiable formulae $\Phi_1^n, \Phi_2^n, \cdots$.

As we have seen in the last section, the main difference between S's output and the view of the verifier is in the y_s's: they are all quadratic residues modulo the corresponding x_s's in S's output, while they are all quadratic nonresidues in $View$. We will now describe an efficient nonuniform algorithm $C = \{C_n\}_{n\in\mathcal{I}}$. Each C_n takes two inputs: $j \geq 0$ and $(x, y) \in PAIR(n) = \{(u, v) : u \in BL(n), v \in J_u^{+1}\}$; and has "wired-in" the formulae $\Phi_1^n, \cdots, \Phi_{R(n)}^n$ along with their lexicographically smaller satisfying assignments. Roughly speaking, C_n produces as output a "random" string and "proofs" for all formulae Φ_i^n's. C_n selects the input pair (x, y) as the jth auxiliary pair. All prior pairs are selected as simulator S does and all subsequent pairs as prover P does. Thus, C_n "knows" the factorization of the Blum modulus for all auxiliary pairs except (x, y). Nonetheless, algorithm C_n will use (x, y) as S would if $y \in QR_x$, and as P would if $y \in NQR_x$. More formally, C_n is designed so as to enjoy the following properties. Set

$$Space(n, j, QR) = \{x \stackrel{R}{\leftarrow} BL(n);\ y \stackrel{R}{\leftarrow} QR_x;\ s \stackrel{R}{\leftarrow} C_n(j, x, y) : s\},$$

$$Space(n,j,NQR) = \{x \stackrel{R}{\leftarrow} BL(n);\ y \stackrel{R}{\leftarrow} NQR_x;\ s \stackrel{R}{\leftarrow} C_n(j,x,y) : s\}.$$

Then,

Property (1) $Space(n,0,NQR) = View(n, \Phi_1^n, \cdots, \Phi_{R(n)}^n)$,

Property (2) $Space(n, nR(n)+1, QR) = \{s \stackrel{R}{\leftarrow} S(1^n, \Phi_1^n, \cdots, \Phi_{R(n)}^n) : s\}$,

Property (3) $Space(n,j,QR) = Space(n,j+1,NQR)$.

From these properties we will conclude that the existence of D violates the QRA. We now formally describe the algorithm, and then prove all the stated properties.

The Algorithm C_n

"C_n has "wired-in" the $R(n)$-tuple $(\Phi_1^n, \cdots, \Phi_{R(n)}^n)$ and, for each $\Phi \in \{\Phi_1^n, \cdots, \Phi_{R(n)}^n\}$, the lexicographically smaller satisfying assignment t_Φ."

Input: "An integer $j \in [0, nR(n)+1]$. A pair $(x,y) \in PAIR(n)$."

1. "Choose ρ and choose and certify first auxiliary pair."
   If $j = 0$ then set $x_0 = x$ and $y_0 = y$.
   Else randomly select two n-bit primes $p_0, q_0 \equiv 3 \bmod 4$, set $x_0 = p_0 q_0$, and select $y_0 \in QR_{x_0}$.
   Execute procedure `Gen_ρ_and_Proof1`(x_0, y_0), thus obtaining ρ and $Proof_0$.
2. "Choose other auxiliary pairs."
   "$Tree$ contains the indices of auxiliary pairs that are used to certify two others auxiliary pairs. $Count$ contains the number of all selected auxiliary pairs."
   Set $Tree$ = empty set and $Count = 1$.
   For each formula $\Phi \in \{\Phi_1^n, \cdots, \Phi_{R(n)}^n\}$ do:
      Set $b_0^\Phi = 0$ and randomly select n bits $b_1^\Phi, \cdots, b_n^\Phi$.
      For $i = 0, \cdots, n$ do:
         Set $s = b_0^\Phi \cdots b_i^\Phi$
         If $s \notin Tree$ then
            Add s to $Tree$. Randomly select four n-bit primes
            $p_{s0}, q_{s0}, p_{s1}, q_{s1} \equiv 3 \bmod 4$.
            "Choose 0-child."
            If $Count = j$ then set $x_{s0} = x$, $y_{s0} = y$.
            If $Count < j$ then set $x_{s0} = p_{s0}q_{s0}$ and randomly select $y_{s0} \in QR_{x_{s0}}$.
            If $Count > j$ then set $x_{s0} = p_{s0}q_{s0}$ and randomly select $y_{s0} \in NQR_{x_{s0}}$.
            $Count = Count + 1$
            "Choose 1-child."
            If $Count = j$ then set $x_{s1} = x$, $y_{s1} = y$.
            If $Count < j$ then set $x_{s1} = p_{s1}q_{s1}$ and randomly select $y_{s1} \in QR_{x_{s1}}$.
            If $Count > j$ then set $x_{s1} = p_{s1}q_{s1}$ and randomly select $y_{s1} \in NQR_{x_{s1}}$.
            $Count = Count + 1$
3. "Choose τ_1 and τ_2."
   Let w be the index of (x,y), that is $(x_w, y_w) = (x,y)$. If there is no such w, set w = empty string.[11]
   If $w \in Tree$ then

[11] It may happen that fewer than j (different) auxiliary pairs will be chosen. To give an extreme example, it may happen that, for all Φ, the bits $b_1^\Phi \cdots b_n^\Phi$ are always the same.

Compute $\Psi_w = CR(x_{w0}, y_{w0}, x_{w1}, y_{w1})$ and a satisfying assignment t_w for Ψ_w.
Execute procedure `Sample_τ_and_Proof2`(Ψ_w, t_w, x_w, y_w) obtaining τ_1 and $Proof\Psi_w$.
Randomly select a $2n^4$-bit string τ_2.
Else, if $w = b_0^\Phi \cdots b_n^\Phi$, for $\Phi \in \{\Phi_1^n, \cdots, \Phi_{R(n)}^n\}$, then
Execute procedure `Sample_τ_and_Proof2`(Φ, t_Φ, x, y) obtaining τ_2 and $Proof\Phi$.
Randomly select a $2n^a$-bit string τ_1.
Else, randomly select a $2n^a$-bit string τ_1 and a $2n^4$-bit string τ_2.
4. "Choose proofs with respect to τ_1 and τ_2."
Set $PROOF$= empty string and $Tree = \{w\}$.
For each formula $\Phi \in \{\Phi_1^n, \cdots, \Phi_{R(n)}^n\}$ do:
4.1. "Certify auxiliary pairs."
Append $(x_0, y_0), Proof_0$, and $b_0^\Phi \cdots b_n^\Phi$ to $PROOF$.
For $i = 0, \cdots, n$ do:
Set $s = b_0^\Phi \cdots b_i^\Phi$.
If $s \notin Tree$ then
Add s to $Tree$.
If $y_s \in NQR_{x_s}$ then
Compute $\Psi_s = CR(x_{s0}, y_{s0}, x_{s1}, y_{s1})$ and a satisfying assignment t_s for Ψ_s.
Execute procedure `Prove`$(\Psi_s, t_s, x_s, y_s, \tau_1)$ obtaining $Proof\Psi_s$.
If $y_s \in QR_{x_s}$ then execute `Gen_Proof2`$(\Psi_s, x_s, y_s, p_s, q_s, \tau_1)$ obtaining $Proof\Psi_s$.
Append (x_{s0}, y_{s0}), (x_{s1}, y_{s1}), and $Proof\Psi_s$ to $PROOF$.
4.2. "Prove Φ."
Set $s = b_0^\Phi \cdots b_n^\Phi$.
If $s \neq w$ then
If $y_s \in NQR_{x_s}$ then execute procedure `Prove`$(\Phi, t_\Phi, x_s, y_s, \tau_2)$ obtaining $Proof\Phi$.
If $y_s \in QR_{x_s}$ then execute `Gen_Proof2`$(\Phi, x_s, y_s, p_s, q_s, \tau_2)$ obtaining $Proof\Phi$.
Append $Proof\Phi$ to $PROOF$.

Output:$(\rho \circ \tau_1 \circ \tau_2, PROOF)$.

First note that $\{C_n\}_{n \in \mathcal{I}}$ is an efficient nonuniform algorithm. All x_s's (except the jth) are selected along with their prime factors and thus all related computations can be performed in expected polynomial time. All operations concerning x and y are simple multiplications and testing of membership in J_x^{+1}. The size of the set $Tree$ is never bigger than $nR(n)$, and thus membership and add operations are easily performed.

The strings τ_1 and τ_2 constructed by C_n are random. Indeed, either they are randomly selected or they are generated by `Sample_τ_Proof2`. The analysis in §5.4 shows that in the latter case the resulting string τ is random.

Proof of Property (1). Assume $j = 0$ and $y \in NQR_x$. All y_s's are quadratic nonresidues in C_n's output. (x, y) is set equal to (x_0, y_0) and used twice: at step 1 to produce ρ and $Proof_0$, and at step 3 to construct $Proof\Psi_0$. Both the strings $Proof_0$ and $Proof\Psi_0$ have the same probability of being chosen as in $View$ when the first

pair is (x_0, y_0). From Lemma 5.3, each string ρ is equally likely to be constructed at step 1. Thus, $Space(n, 0, NQR) = View(n, \Phi_1^n, \cdots, \Phi_{R(n)}^n)$.

Proof of Property (2). Suppose $j = nR(n) + 1$. To prove $R(n)$ formulae, at most $nR(n)$ auxiliary pairs are needed. Thus, each y_s constructed by C_n belongs to QR_{x_s}. All the strings $Proof\Psi_s$'s and $Proof\Phi$'s are constructed in exactly the same way, both by S and by C_n. Hence, $Space(n, nR(n) + 1, QR) = \{s \stackrel{R}{\leftarrow} S(1^n, \Phi_1^n, \cdots, \Phi_{R(n)}^n) : s\}$

Proof of Property (3). Consider now the two probability spaces $Space(n, j, QR)$ and $Space(n, j + 1, NQR)$. In both spaces the auxiliary pairs are randomly chosen so that the first j y_s's are quadratic residues modulo the corresponding x_s's and, from the $(j + 1)$st on, all the y_s's are quadratic nonresidues. All computations concerning pairs (x_s, y_s) different from (x, y) are performed in the same way. The pair (x, y) is used to construct either a proof $Proof\Psi_s$ for a formula Ψ_s derived from a reduction or a proof $Proof\Phi$ for one of the formulae Φ_i^n, or is never used. In the former two cases the proof is generated using the procedure `Sample_τ_and_Proof2`. When $y \in NQR_x$ ($y \in QR_x$), this procedure returns a string $Proof$ that has the same distribution as if it where generated by the procedure `Prove` (`Gen_Proof2`). Thus, $Space(n, j, QR) = Space(n, j + 1, NQR)$.

We now conclude the proof of Theorem 6.3. We have assumed that D distinguishes between $S(1^n, \Phi_1^n, \cdots, \Phi_{R(n)}^n)$'s output and $View(n, \Phi_1^n, \cdots, \Phi_{R(n)}^n)$. From properties (1) and (2), then, this is tantamount to saying that D distinguishes between $Space(n, 0, NQR)$ and $Space(n, nR(n) + 1, QR)$. By the pigeon-hole principle, and because of Property (3), for all $n \in \mathcal{I}$ there exists $j = j(n)$, $0 \le j \le nR(n) + 1$, such that D distinguishes between $Space(n, j, QR)$ and $Space(n, j, NQR)$. That is, for all $n \in \mathcal{I}$,

$$\left|P_j(n, QR) - P_j(n, NQR)\right| \ge 1/((nR(n) + 2)n^d)$$

where $P_j(n, QR) = Pr(s \stackrel{R}{\leftarrow} Space(n, j, QR) : D_n(s) = 1)$ and $P_j(n, NQR) = Pr(s \stackrel{R}{\leftarrow} Space(n, j, NQR) : D_n(s) = 1)$. Thus, composing each $C_n(j(n), \cdot, \cdot)$ with D_n, one obtains an efficient nonuniform algorithm that violates the QRA. □

6.5. Proving theorems of arbitrary size. Given a reference string of $8n^3 + 2n^a + 2n^4$ bit, the proof system (P, V) of §6.1 can be used to prove in zero-knowledge the 3-satisfiability of an arbitrary number of 3-satisfiable formulae, but each of them must have at most n clauses. However, the same proof system can be used to prove 3-satisfiable formulae with any number of clauses. The idea is perhaps best conveyed in an informal manner. Given a formula Φ with k clauses, the prover computes a certified auxiliary pair (x^Φ, y^Φ) and the lexicographically smaller satisfying assignment t for Φ. To label each literal u_j of Φ the prover randomly selects $r_j \in Z_{x^\Phi}^*$ and, if $t(u_j) = 1$ he associates with u_j the label $w_j = r_j^2 y^\Phi \bmod x^\Phi$; otherwise the label $w_j = r_j^2 \bmod x^\Phi$. The label associated with $\overline{u}_j$ is $w_j y^\Phi \bmod x^\Phi$. Essentially, a literal has an element in NQR_{x^Φ} as label if and only if it is made true by t. To prove that $\Phi \in 3SAT$, the prover proves that each clause has at least an element of NQR_{x^Φ} among the labels of its three literals. That is, consider the language $L = \{(y_1, y_2, y_3, x)$: at least one of y_1, y_2, y_3 belongs to $NQR_x\}$. Then $L \in NP$ and therefore there exists a fixed polynomial-time computable reduction RED such that

$$\Phi' = RED(y_1, y_2, y_3, x) \in 3SAT_{n^f} \iff (y_1, y_2, y_3, x) \in L,$$

where f is a fixed constant depending only on RED. Therefore, to prove that the ith clause is satisfied, the prover computes the formula Φ_i using the reduction RED and

proves that $\Phi_i \in 3SAT$. By the property of the reduction the length of the formula is upper bounded by n^f and can thus be proved 3-satisfiable using the previously described proof system (P, V) with a reference string of $8n^{3f} + 2n^{sf} + 2n^{4f}$ bits. Therefore, we have reduced the problem of proving the 3-satisfiability of one formula with many clauses to that of proving the 3-satisfiability of many formulae, each with at most n^f clauses.

6.6. Efficient provers. In the proof system of §6.1, for convenience of presentation, the prover P was made quite powerful. For instance, P needs to find the lexicographically first satisfying assignment of a formula for proving that it is in 3SAT. This, however, is not necessary. It is easily seen that, under the QRA, the verifier would obtain an undistinguishable view [GoMiRa], no matter which satisfying assignment the prover may use. Also, it is possible for the prover to have access to a random oracle instead of a random selector and still generate essentially the same view to a polynomial-time verifier. In fact, by well-known techniques, a random oracle can be transformed to a random function associating each string with σ a "polynomially longer" random string. This random string may be used to select the necessary primes and quadratic residues and nonresidues with essentially the same odds as for a random selector. Actually, if one replaces a random oracle with a polyrandom function as in Goldreich, Goldwasser, and Micali [GoGoMi], the view of the verifier would still be indistinguishable from the one it obtains from P. These functions exist under the QRA[12] and the replacement only entails that the same short, randomly selected string should be remembered throughout the proving process.

In sum, the prover may very well be polynomial time, as long as it is given satisfying assignments for the formulae that need to be proved satisfiable in noninteractive zero knowledge.

This is an important point, and can be shown to hold not only for our specific noninteractive ZKPS, but also for any other that shares our algorithmic structure. Since, however, systems with a different structure and relying on weaker intractability assumptions have already been found (see below), we decline to formalize this point in our paper. Our goal, at this point, is making precise the notion of noninteractive zero-knowledge and showing its feasibility.

7. Recent improvements and related works. Two main open problems were posed in [DeMiPe1], namely,

1. whether many provers could share the same random string and[13]
2. whether it is possible to implement noninteractive zero-knowledge with a general complexity assumption, rather than on our specific number-theoretic one.

Recently, both our questions have been solved in a beautiful paper by Feige, Lapidot, and Shamir [FeLaSh]. They show that any number of provers can share the same random string and that any trap-door permutation can be used instead of quadratic residuosity. They also show that one-way permutations are sufficient for bounded noninteractive zero-knowledge, but the prover *needs* to have exponential computing

[12] In fact Blum, Blum, and Shub [BlBlSh] show that the QRA implies the existence of a polyrandom generator in the sense of Blum and Micali [BlMi] and Yao [Ya], and [GoGoMi] show that any polyrandom generator can be used to construct a polyrandom function.

[13] Indeed, if this had been done in our protocol, completeness and soundness would still hold. However, it is not clear that the zero-knowledge would be preserved. Without changing our proof systems, we can handle only a moderate number of provers. This number is limited for the same reasons outlined in footnote 6.

power. Our first question was also independently solved by De Santis and Yung [DeYu].

Noninteractive zero-knowledge has been shown to yield a new paradigm for digital signature schemes by Bellare and Goldwasser [BeGo].

De Santis, Micali, and Persiano [DeMiPe2] show that, if any one-way function exists, after an interactive preprocessing stage, any "sufficiently short" theorem can be proven noninteractively and in zero-knowledge.

Kilian, Micali, and Ostrovsky [KiMiOs] have shown that, if any one-way function exists, after a preprocessing stage consisting of a "few" executions of an oblivious transfer protocol, any theorem can be proven in zero knowledge and noninteractively. (Namely, after executing $O(k)$ oblivious transfers, the probability of accepting a false theorem is 1 in 2^k.) Bellare and Micali [BeMi] show that, based on a complexity assumption, it is possible to build public-key cryptosystems in which oblivious transfer is itself implementable without any interaction.

8. A general open problem. An obvious open problem in noninteractive zero-knowledge consists of finding more efficient proof systems. However, in our opinion, a more important one is decreasing the needed complexity assumption. This effort should be extended to all of cryptography at this point in its development.

Introducing new cryptographic primitives is crucial, but would be essentially impossible without first relying on some special, though hopefully well studied, complexity assumptions. It is important, though, to later find the minimal assumptions for implementing these primitives. In fact, "extra structure" may make proving that the desired property holds easier, but may also force the underlying complexity assumption to be false. Personally, Micali finds a dramatic difference between one-way functions and one-way permutations. (Breaking a glass is quite easy. Putting it back together is certainly harder, but what if we were guaranteed that there is a unique way to do so?)

We believe noninteractive zero-knowledge to be a fundamental primitive, one deserving the effort to establish the minimal assumptions needed for it to be securely implemented. We thus hope the following question will be settled: If one-way functions exist, does 3SAT have noninteractive zero-knowledge proof systems whose prover, given the proper witness, needs only to work in polynomial time?

9. Acknowledgments. We wholeheartedly thank Mihir Bellare for his constructive and generous criticism throughout this research.

Many thanks to Shafi Goldwasser and Mike Sipser for their encouraging support, technical and spiritual. (Make it also financial next time!)

Given that our work improves on that of [BlFeMi], we regret that we could not collaborate or reach Paul Feldman. We certainly would have benefited from his insights.

Finally, hoping that he or she accepts direct compliments, thanks to an anonymous referee for very intelligent comments.

REFERENCES

[AdHu] L. M. ADLEMAN AND M. A. HUANG, *Recognizing primes in random polynomial time*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 462–470.

[An] D. ANGLUIN, *Lecture notes on the complexity of some problems in number theory*, Tech. Report 243, Yale University, Dept. of Computer Science, New Haven, CT, 1982.

[AnVa] D. ANGLUIN AND L. VALIANT, *Fast probabilistic algorithms for Hamiltonian circuits and matchings*, J. Comput. System Sci., 18 (1979), pp. 155–193.

[Ba] L. BABAI, *Trading group theory for randomness*, in Proc. 17th Symposium on Theory of Computing, Providence, RI, 1985, pp. 421–429.

[BaMo] L. BABAI AND S. MORAN, *Arthur–Merlin games: A randomized proof system and a hierarchy of complexity classes*, J. Comput. System Sci., 36, (1988), pp. 254–276.

[BeGo] M. BELLARE AND S. GOLDWASSER, *New paradigm for digital signature and message authentication based on non-interactive zero-knowledge proofs*, in Advances in Cryptology–CRYPTO 89, Lecture Notes in Computer Science, 435, Springer–Verlag, Berlin, New York, 1989, pp. 194–211.

[BeGoWi] M. BEN-OR, S. GOLDWASSER, AND A. WIGDERSON, *Completeness theorems for non-cryptographic fault-tolerant distributed computations*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 1–10.

[BeMi] M. BELLARE AND S. MICALI, *Non-interactive oblivious transfer and applications*, in Advances in Cryptology–CRYPTO 89, Lecture Notes in Computer Science, 435, Springer–Verlag, Berlin, New York, 1989, pp. 547–559.

[BeMiOs] M. BELLARE, S. MICALI, AND R. OSTROWSKY, *Perfect zero-knowledge in constant rounds*, in Proc. 22nd Annual ACM Symposium on the Theory of Computing, Baltimore, MD, 1990, pp. 482–493.

[BlBlSh] M. BLUM, L. BLUM, AND M. SHUB, *A simple and secure pseudo-random number generator*, SIAM J. Comput., 15 (1986), pp. 364–383.

[BlFeMi] M. BLUM, P. FELDMAN, AND S. MICALI, *Non-interactive zero-knowledge proof systems and applications*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 103–112.

[BlMi] M. BLUM AND S. MICALI, *How to generate cryptographically strong sequence of pseudo-random bits*, SIAM J. Comput., 13 (1984), pp. 850–864.

[Bl1] M. BLUM, *Coin flipping by telephone*, IEEE COMPCON, High Technology in the Information Age, Spring 1982, pp. 133–137.

[Bl2] ———, *How to prove a theorem so no one else can claim it*, in Proc. Internat. Congress of Mathematicians, Berkeley, CA, 1986, pp. 1444–1451.

[BoHaZa] R. BOPPANA, J. HASTAD, AND S. ZACHOS, *Does co-NP have short interactive proofs?*, Inform. Process. Lett., 25 (1987), pp. 127–132.

[ChCrDa] D. CHAUM, C. CREPAU, AND I. DAMGÄRD, *Multiparty unconditionally secure protocols*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 11–19.

[Co] S. A. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, New York, NY, pp. 151–158.

[DeMiPe1] A. DE SANTIS, S. MICALI, AND G. PERSIANO, *Non-interactive zero-knowledge proof-systems*, in Advances in Cryptology–CRYPTO 87, Lecture Notes in Computer Science, 293 Springer–Verlag, Berlin, New York, 1987, pp. 52–72.

[DeMiPe2] A. DE SANTIS, S. MICALI, AND G. PERSIANO, *Non-interactive zero-knowledge proof-systems with preprocessing*, in Advances in Cryptology–CRYPTO 88, of Lecture Notes in Computer Science, 403, Springer–Verlag, Berlin, New York, 1988, pp. 269–283.

[DeYu] A. DE SANTIS AND M. YUNG, *Cryptograhic applications of the non-interactive metaproof and many-prover systems*, in Advances in Cryptology–Crypto 90, Springer–Verlag, Berlin, New York.

[ErSp] P. ERDOS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[FeFiSh] U. FEIGE, A. FIAT, AND A. SHAMIR, *Zero-knowledge proofs of identity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 210–217.

[FeLaSh] U. FEIGE, A. LAPIDOT, AND A. SHAMIR, *Multiple non-interactive zero-knowledge proofs based on a single random string*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 308–317.

[Fo] L. FORTNOW, *The complexity of perfect zero-knowledge*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 204–209.

[FuGoMaSiZa] M. FURER, O. GOLDREICH, Y. MANSOUR, M. SIPSER, AND S. ZACHOS, *On completeness and soundness in interactive proof systems*, in Advances in Computing Research, Vol. 5. Randomness and Computation, S. Micali, ed., JAI Press Inc., Greenwich, CT, pp. 429–442.

[GaJo] M. GAREY AND D. JOHNSON, *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman, New York, 1979.

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

[GoKi] S. Goldwasser and J. Kilian, *Almost all primes can be quickly certified*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 316–329.

[GoMi1] S. Goldwasser and S. Micali, *Probabilistic Encryption*, J. Comput. System Sci., 28 (1984), pp. 270–299.

[GoMi2] ———, *Proofs with untrusted oracles*, manuscript.

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

[GoMiRi] S. Goldwasser, S. Micali, and R. Rivest, *A digital signature scheme secure against adaptive chosen-message attack*, SIAM J. Comput., 17 (1988), pp. 281–308.

[GoMiWi1] O. Goldreich, S. Micali, and A. Wigderson, *Proofs that yield nothing but their validity and a methodology of cryptographic design*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, 1986, pp. 174–187.

[GoMiWi2] ———, *How to play any mental game*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 218–229.

[GoSi] S. Goldwasser and M. Sipser, *Private coins versus public coins in interactive proof-systems*, in Proc. 18th Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 59–68.

[KaLi] R. Karp and R. Lipton, *Turing machines that take advice*, L'Enseignement Mathematique, 28, pp. 191–209, STOC 1980.

[KiMiOs] J. Kilian, S. Micali, and R. Ostrowsky, *Minimum resource zero-knowledge*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, Reserach Triangle Park, NC, 1989, pp. 474–479.

[MiSh] S. Micali and A. Shamir, *An improvement of the Fiat–Shamir identification and signature scheme*, Crypto 88, Lecture Notes in Computer Science 403, Springer–Verlag, Berlin, New York, 1988.

[NaYu] M. Naor and M. Yung, *Public-key cryptosystems provably secure against chosen cyphertext attack*, in Proc. 22nd Symposium on Theory of Computing, Baltimore, MD, 1990, pp. 427–437.

[NiZu] I. Niven and H. S. Zuckerman, *An introduction to the theory of numbers*, John Wiley, New York, 1960.

[Ra1] M. Rabin, *Probabilistic algorithm for testing primality*, J. Number Theory, 12 (1980), pp. 128–138.

[Ra2] ———, *Digitalized signatures and public-key functions as intractable as factorization*, Tech. Report MIT/LCS/TR-212, MIT Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA, 1978.

[Ra3] ———, *Transaction protection by beacons*, Tech. Report 29-81, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1981.

[Sh] D. Shanks, *Solved and unsolved problems in number theory*, Chelsea, New York, 1976.

[SoSt] R. Solovay and V. Strassen, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85

[Ya] A. Yao, *Theory and applications of trapdoor functions*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, Chicago, IL, 1982, pp. 80–91.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1119–1127, December 1991
© 1991 Society for Industrial and Applied Mathematics
008

ELIMINATION OF INFREQUENT VARIABLES IMPROVES AVERAGE CASE PERFORMANCE OF SATISFIABILITY ALGORITHMS*

JOHN FRANCO†

Abstract. Preprocessing a random instance I of CNF Satisfiability in order to remove infrequent variables (those which appear once or twice in an instance) from I is considered. The model used to generate random instances is the popular random-clause-size model with parameters n; the number of clauses, r; the number of Boolean variables from which clauses are composed; and p, the probability that a variable appears in a clause as a positive (or negative) literal. It is shown that exhaustive search over such preprocessed instances runs in polynomial average time over a significantly larger parameter space than has been shown for any other algorithm under the random-clause-size model when $n = r^\epsilon$, $\epsilon < 1$, and $pr < \sqrt{\epsilon r \ln(r)}$. Specifically, the results are that random instances of Satisfiability are "easy" in the average case if $n = r^\epsilon$, $\frac{2}{3} > \epsilon > 0$, and $pr < (\ln(n)/4)^{1/3} r^{2/3-\epsilon}$; or $n = r^\epsilon$, $1 > \epsilon \geq \frac{2}{3}$, $pr < (1-\epsilon-\delta)\ln(n)/\epsilon$ for any $\delta > 0$; or $pn \to 0$, $pr < \gamma \ln\ln(n)$ for any $\gamma > 0$.

Key words. Satisfiability, NP-complete, probabilistic analysis, resolution

AMS(MOS) subject classification. 68B10

1. Introduction. The Satisfiability problem is to determine whether there exists a truth assignment to the variables of a given CNF Boolean expression which causes it to have value *true*. If such a truth assignment exists, we say the expression is satisfiable; otherwise it is unsatisfiable. The problem is NP-complete so there is no known polynomial-time algorithm for solving it. Several papers have been concerned with the analysis of algorithms for Satisfiability that run in polynomial average time. These results depend on an assumed probabilistic input model. One popular model is the "random-clause-size" model, which we refer to as $M(n, r, p)$.

Let $L = \{v_1, \bar{v}_1, v_2, \bar{v}_2, \cdots, v_r, \bar{v}_r\}$ be a set of $2r$ literals. According to the model $M(n, r, p)$, n disjunctions (called clauses) are generated as follows: for each clause C_i, for all literals $l \in L$, put l in C_i with probability p, independently of the placement of other literals and clauses. Note that it is possible for a pair of complementary literals (associated with the same variable) to be present in a clause. It is also possible to generate an empty (or null) clause using this model. If an instance contains a null clause, it is trivially unsatisfiable. The preponderance or absence of null clauses in random instances is controlled by the product pr, which is half the average number of literals in a clause. From [2] a random instance possesses a null clause with probability tending to 1 if the product $pr < \ln(n)/2$.

In the literature, polynomial average time results for Satisfiability algorithms are known only if $n = r^\epsilon$, $1 > \epsilon > 0$, $pr < \sqrt{\ln(n)} \cdot r^{1/2-\epsilon}$ [4]; or $n = r^\gamma$, $\gamma > 1$, $pr < (\gamma - 1)\ln(n)/(2\gamma)$ [5]; or $n = \beta r$, β a positive constant, and $pr < f(\beta)$, where f is some complicated function of β; or $pr > \sqrt{r \ln(n)}$ [3]. This space of parameters is depicted as region **I** in Fig. 1 (the scale of the figure is such that factors of $\ln(n)$ are not distinguished). Furthermore, no polynomial average time results are published for parameters set in region **II** or region **III** of Fig. 1. Also of interest is a result in [1]

* Received by the editors October 16, 1989; accepted for publication (in revised form) February 7, 1990. This work was carried out in part at the FAW, Helmholtzstraße 16, D-7900 Ulm/Donau, Germany, and is based on research supported in part by the Air Force Office of Scientific Research grant AFOSR 84-0372 and 89-0186.

† Department of Computer Science, University of Cincinnati, Cincinnati, Ohio 45221-0008 (john.franco@uc.edu or franco@ucunix.san.uc.edu).

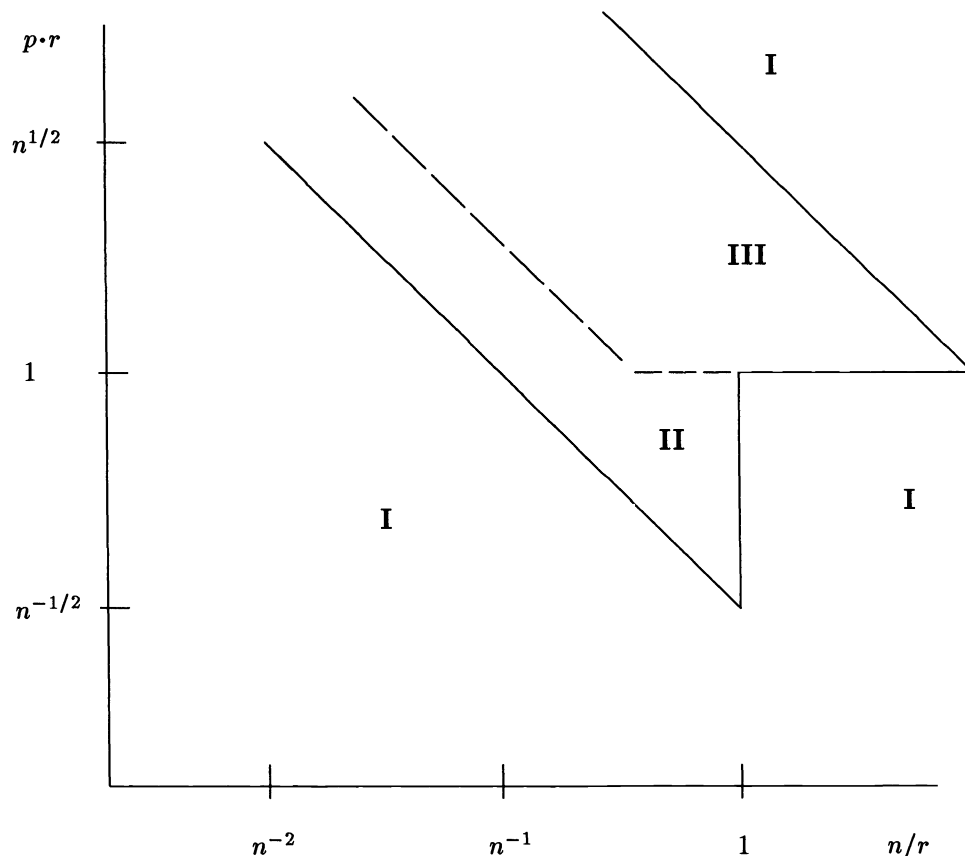


FIG. 1. *Regions of the parameter space of model $M(n, r, p)$. Random instances generated using parameters which fall into region* **I** *are solved in polynomial time by some previously analyzed algorithm. The algorithm PLR requires superpolynomial average time for parameters set in regions* **II** *and* **III**. *Algorithm INFREQ solves random instances in polynomial average time in region* **II**. *No known algorithm takes polynomial average time for parameters set in region* **III**.

which shows that an algorithm based on the pure literal rule, called *PLR*, requires superpolynomial average time if $n = r^{\epsilon}$, $1 > \epsilon > \frac{1}{2}$, and $pr > \sqrt{\omega(r)\ln(n)} \cdot r^{1/2-\epsilon}$, where $\omega(r)$ is any growing function of r. In Fig. 1, this range of parameters includes region **II**. Thus, there is a significant range of pr for which null clauses exist in random instances with high probability but no published polynomial average time analysis exists and at least one nontrivial algorithm, namely, *PLR*, requires superpolynomial average time. This range is depicted in Fig. 1 as the part of region **II** below the extension of the lower boundary between regions **I** and **III**.

In this paper we extend the parameter space over which polynomial average time results are known. We present an algorithm called *INFREQ* and show that it has polynomial average time performance over a range of parameter values including region **II** of Fig. 1; no published analysis has shown this region covered by a polynomial average time algorithm.

INFREQ uses substitution rules to eliminate or combine clauses containing infrequent variables: that is, variables occurring only once or twice in an instance. Infrequent variables are also eliminated by applying these rules. In addition, *INFREQ* checks the input for a null clause before processing. If one is found, *INFREQ* immediately stops with the result that the input is unsatisfiable. Otherwise, *INFREQ* does an exhaustive search over all remaining variables for a solution.

The results of this paper show that exhaustive search over the variables which are not infrequent is, for certain relationships between the parameters p, n, and r, speeded up considerably as a result of the null check and the preprocessing. The idea seems to be generalizable and may represent the first of a family of such results that will take care of a large portion of the remaining parameter space for which polynomial average time results are not now known. The result of such a generalization, to the extent that it is possible, is apparent from the analysis presented here.

Specifically, the results of this paper are that random instances of Satisfiability are "easy" in the average case if $n = r^{\epsilon}$, $\frac{2}{3} > \epsilon > 0$, and $pr < (\ln(n)/4)^{1/3} r^{2/3-\epsilon}$; or $n = r^{\epsilon}$, $1 > \epsilon \geq \frac{2}{3}$, and $pr \leq (1 - \epsilon - \delta)\ln(n)/\epsilon$ for any $\delta > 0$; or $pn \to 0$, and $pr < \gamma \ln\ln(n)$ for any $\gamma > 0$. These results include region II in Fig. 1, a region not covered by a published polynomial average time result. The first of these results is due to the resolution component and does not depend on the presence of null clauses in an instance. Thus, in Fig. 1, the left boundary of region **III** is due to limited resolution. As will be explained in the remarks following Theorem 3.1, random instances generated according to the parameter space of the first result have relatively few variables that are not infrequent. Therefore, the exhaustive search in *INFREQ* is over a sufficiently small number of variables to obtain polynomial average time. The second result depends on checking for null clauses *and* the elimination of infrequent variables. In Fig. 1, the portion of region **II** that is below an imaginary extension of the lower boundary of region **III** is due to this effect. In this case the average number of variables that are not infrequent is considerable, but *INFREQ* has polynomial average time because of the combination of null clause check and exhaustive search.

2. The algorithm. For convenience, we write a clause as a tuple of the literals it contains. For example, the clause $(x \vee y \vee z)$ is written as (x, y, z). Similarly, we write the conjunction of two clauses $C_1 \wedge C_2$ as C_1, C_2.

Let a variable which appears exactly once in an instance I be called a *unit* variable. Let a variable which appears exactly twice in I be called a *double* variable. Let a variable which appears at least three times in I be called a *serious* variable. Table 1 defines substitutions for clauses in I containing unit and double variables. In the table we use v to denote a positive literal taken from a unit or double variable, $\bar{v}$ a negative literal so taken, and x and y either a positive or negative literal which is not necessarily taken from a unit or double variable.

When we say *apply unit elimination* we mean, according to Table 1, look for a clause containing a unit variable v and replace it with the logical value *true*; if no such clause exists, do nothing. Similar statements hold for applying any of the other substitution rules listed in the table. It is possible that, after repeated applications of double-variable substitution rules, some double variables will occur only once in I. By *clean up double variables* we mean eliminate all clauses containing double variables that appear once in I.

Consider the following algorithm for solving instances of Satisfiability.

TABLE 1

Var type	Substitution name	Occurrence	Replacement
unit	unit elimination	$(v, x, \cdots)$	*true*
		$(\bar{v}, x, \cdots)$	
double	double elimination	$(v, v, x, \cdots)$	
		$(\bar{v}, \bar{v}, x, \cdots)$	
	trivial elimination	$(v, \bar{v}, x, \cdots)$	
	pure literal rule	$(v, x, \cdots), (v, y, \cdots)$	
		$(\bar{v}, x, \cdots), (\bar{v}, y, \cdots)$	
	resolution	$(v, x, \cdots), (\bar{v}, y, \cdots)$	$(x, \cdots, y, \cdots)$

INFREQ(I):

1. If I has a null clause then return "unsatisfiable"
2. Otherwise,
   a. repeatedly apply double variable substitution rules in order until opportunities vanish
   b. clean up all remaining double variables
   c. repeatedly apply unit elimination until opportunities vanish
   d. for all truth assignments t to serious variables in I, if t satisfies I then return "satisfiable"
3. Return "unsatisfiable"

In step (2d) *INFREQ* terminates as soon as the first satisfiable truth assignment is discovered. It should be apparent that the size of I is not increased by the application of *INFREQ* to I. It should also be apparent that all unit and double variables are eliminated from I in steps (2a), (2b), and (2c) of *INFREQ* (these are the preprocessing steps). Thus, in step (2d), the truth assignment t is an assignment to all variables which appear in the processed I.

LEMMA 2.1. *INFREQ returns "satisfiable" if and only if I is satisfiable.*

Proof. The proof is straightforward and is omitted. □

3. The analysis. To simplify the analysis, we show that the expected number of iterations in step (2d) of *INFREQ* is bounded by a polynomial in n under several conditions. Since the complexity of each step is polynomially bounded, the average running time of *INFREQ* must then be polynomially bounded under those conditions as well.

Let $I_=(y)$ denote the event that the input contains exactly y serious variables. Let $I_\geq(y)$ denote the event that the input contains at least y serious variables. Let I_ϕ denote the event that the input contains a null clause. Let $T(n, r, p)$ denote the average number of steps executed by *INFREQ* given that instances are generated according to model $M(n, r, p)$. Then, since the number of steps required by exhaustive search on an input with exactly y serious variables is at most 2^y, we can write

$$T(n,r,p) \leq Pr(I_\phi) + \sum_{y=1}^{r} 2^y \cdot Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + 2Pr(\bar{I}_\phi \wedge I_=(1)) + \sum_{y=2}^{r} 2^y \cdot Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + 2Pr(\bar{I}_\phi \wedge I_=(1)) + \sum_{y=2}^{r}\left(1 + \sum_{x=0}^{y-1} 2^x\right) Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + 2Pr(\bar{I}_\phi \wedge I_=(1)) + \sum_{y=2}^{r} Pr(\bar{I}_\phi \wedge I_=(y)) + \sum_{y=2}^{r}\sum_{x=0}^{y-1} 2^x \cdot Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + Pr(\bar{I}_\phi \wedge I_=(1)) + Pr(\bar{I}_\phi \wedge I_\geq(1)) + \sum_{y=2}^{r}\sum_{x=0}^{y-1} 2^x \cdot Pr(\bar{I}_\phi \wedge I_=(y)).$$

Interchanging the order of summation in the double sum gives

$$T(n,r,p) \leq Pr(I_\phi) + Pr(\bar{I}_\phi \wedge I_=(1)) + Pr(\bar{I}_\phi \wedge I_\geq(1))$$
$$+ \sum_{x=1}^{r-1}\sum_{y=x+1}^{r} 2^x \cdot Pr(\bar{I}_\phi \wedge I_=(y)) + \sum_{y=2}^{r} Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + Pr(\bar{I}_\phi \wedge I_=(1)) + \sum_{y=2}^{r} Pr(\bar{I}_\phi \wedge I_=(y)) + Pr(\bar{I}_\phi \wedge I_\geq(1))$$
$$+ \sum_{x=1}^{r-1} 2^x \sum_{y=x+1}^{r} Pr(\bar{I}_\phi \wedge I_=(y))$$

$$= Pr(I_\phi) + 2Pr(\bar{I}_\phi \wedge I_\geq(1)) + \sum_{x=1}^{r-1} 2^x \cdot Pr(\bar{I}_\phi \wedge I_\geq(x+1))$$

$$\leq 1 + 2Pr(\bar{I}_\phi \wedge I_\geq(1)) + \sum_{x=1}^{r-1} 2^x \cdot Pr(\bar{I}_\phi \wedge I_\geq(x+1))$$

$$\leq 3 + \sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x \cdot Pr(\bar{I}_\phi) + \sum_{x=\lfloor 6\mu \rfloor+1}^{r-1} 2^x \cdot Pr(I_\geq(x+1))$$

$$(1) \qquad = 3 + \sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x \cdot Pr(\bar{I}_\phi) + \sum_{x=\lfloor 6\mu \rfloor+2}^{r} 2^{x-1} \cdot Pr(I_\geq(x)),$$

where μ is the mean number of serious variables in an instance. The appearance of the number 6 in (1) will be explained below.

First, we obtain a bound on the second sum in (1). Since variables are placed independently in clauses, the number of serious variables in an instance is binomially distributed. By the Chernoff bound for binomial distributions [6],

$$Pr(I_\geq((1+\beta)\mu)) < e^{-\ln(1+\frac{\beta}{1-p})\frac{\beta\mu}{2}},$$

$\beta > 0$. Below we shall make use of this and the easily verified fact that $x\ln(2) - \ln(\frac{x}{\mu})(\frac{x}{\mu}-1)\frac{\mu}{2} < 0$ if $x \geq \lceil 6\mu \rceil + 1$ and $p > 0$ (this is the reason why 6 appears in (1)). Thus,

$$\sum_{x=\lfloor 6\mu \rfloor+2}^{r} 2^{x-1} \cdot Pr(I_\geq(x)) \leq \sum_{x=\lfloor 6\mu \rfloor+2}^{r} 2^x e^{-\ln\left(\frac{\frac{x}{\mu}-p}{1-p}\right)(\frac{x}{\mu}-1)\frac{\mu}{2}}$$

$$= \sum_{x=\lceil 6\mu \rceil+1}^{r} e^{-\ln(\frac{x}{\mu})(\frac{x}{\mu}-1)\frac{\mu}{2}+x\ln(2)}$$

$$\leq \sum_{x=\lceil 6\mu \rceil+1}^{r} 1 \leq r.$$

Next, we obtain an upper bound on the first sum in (1). The probability that a clause is null is $(1-p)^{2r}$. Hence, the probability that all clauses are not null is $Pr(\bar{I}_\phi) = (1-(1-p)^{2r})^n$. Thus, we write

$$\sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x \cdot Pr(\bar{I}_\phi) = Pr(\bar{I}_\phi) \sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x = (1-(1-p)^{2r})^n \sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x. \tag{2}$$

It may be verified that $(1-p) \geq e^{-p-p^2}$ if $0 \leq p < \frac{1}{2}$. Hence

$$\begin{aligned} \sum_{x=1}^{\lfloor 6\mu \rfloor} 2^x \cdot Pr(\bar{I}_\phi) &\leq (1-(1-p)^{2r})^n 2^{\lfloor 6\mu \rfloor+1} \\ &\leq (1-e^{-2rp(1+p)})^n 2^{\lfloor 6\mu \rfloor+1} \leq e^{-ne^{-2p(1+p)r}} 2^{6\mu+1} \\ &= e^{-ne^{-2p(1+p)r}+\ln(2)(6\mu+1)}. \end{aligned} \tag{3}$$

Now, we compute μ and obtain upper bounds on (3). The probability that a variable is not in a particular clause is the probability that neither literal associated with the variable is in that clause and is equal to $(1-p)^2$. Since clauses are independently chosen, the probability that a variable is not in a given instance is $(1-p)^{2n}$, the probability that a variable appears once in an instance is $2pn(1-p)^{2n-1}$, and the probability that a variable appears twice in an instance is $\binom{2n}{2}p^2(1-p)^{2n-2}$. Therefore, the probability that a variable is a serious variable is

$$1-(1-p)^{2n} - 2pn(1-p)^{2n-1} - n(2n-1)p^2(1-p)^{2n-2},$$

which may be reduced to

$$1-(1-p)^{2n}(1+2pn/(1-p)+2(pn)^2(1-1/n)/(1-p)^2).$$

THEOREM 3.1. *INFREQ runs in polynomial average time if* $n = r^\epsilon$, $\epsilon \leq \frac{2}{3}$, *and* $pr \leq (\ln(n)/4)^{1/3} r^{2/3-\epsilon}$; *or if* $n = r^\epsilon$, $1 > \epsilon > \frac{2}{3}$, *and* $pr < (1-\epsilon-\delta)\ln(n)/\epsilon$, *for any* $\delta > 0$; *or if* $pn \to 0$ *and* $pr < \gamma \ln\ln(n)$ *for any* $\gamma > 0$.

Proof. If $n = r^\epsilon$, $\epsilon < \frac{2}{3}$, and $pr \leq (\ln(n)/4)^{1/3} r^{2/3-\epsilon}$, then

$$pn = pr \cdot r^{\epsilon-1} \leq (\ln(n)/4)^{1/3} r^{2/3-\epsilon+\epsilon-1} = (\ln(n)/4)^{1/3} r^{-1/3} \to 0.$$

If $1 > \epsilon \geq \frac{2}{3}$ and $pr \leq \ln(n)$, then $pn = pr \cdot r^{\epsilon-1} \leq \epsilon \ln(r) r^{\epsilon-1} \to 0$. So, we assume $pn \to 0$. This implies $p < \frac{1}{2}$. Then the probability that a variable is serious is

$$1-(1-p)^{2n}(1+2pn/(1-p)+2(pn)^2(1-1/n)/(1-p)^2) = (\tfrac{4}{3})(np)^3 + O((np)^4).$$

From now on, we ignore the small term for simplicity. Since variables are placed independently in clauses, the number of serious variables in an instance is binomially distributed with parameters r and $(\frac{4}{3})(np)^3$. Thus, the mean number of serious variables in an instance is $\mu = (\frac{4}{3})(np)^3 r$. Substituting into (3) gives

$$e^{-ne^{-2p(1+p)r}+\ln(2)(8(np)^3 r+1)},$$

which is polynomially bounded if $e^{-ne^{-2p(1+p)r}+\ln(2)(8(np)^3r)}$ is. Therefore, we require

$$-e^{-2p(1+p)r}+5.545(pn)^2(pr)\le \ln(n)/n. \tag{4}$$

Let $n=r^\epsilon$, $1>\epsilon>0$. Then, after rearranging, (4) becomes

$$\begin{aligned} 5.545(pr)^3r^{2(\epsilon-1)} &\le e^{-2pr(1+p)}+\epsilon r^{-\epsilon}\ln(r) \iff \\ 5.545(pr)^3r^{3\epsilon-2} &\le r^\epsilon e^{-2pr(1+p)}+\epsilon\ln(r) \end{aligned} \tag{5}$$

Let $\epsilon\le\frac{2}{3}$ and $pr=(\ln(n)/4)^{1/3}r^\alpha$, α a constant. Then (5) becomes

$$5.545\ln(n)r^{3\alpha+3\epsilon-2}/4\le r^\epsilon e^{-2r^\alpha-2r^{2\alpha-1}}+\epsilon\ln(r). \tag{6}$$

Clearly, (6) is satisfied if $3\alpha+3\epsilon-2\le 0$ or $\alpha\le\frac{2}{3}-\epsilon$. Thus, if $\epsilon\le\frac{2}{3}$ and $pr<(\ln(n)/4)^{1/3}r^{2/3-\epsilon}$, then (2) is polynomially bounded.

Now let $1>\epsilon>2/3$ and $pr=\alpha\ln(n)=\alpha\epsilon\ln(r)$. Then (5) becomes

$$5.545(\alpha\ln(r))^3r^{3\epsilon-2}\le r^{\epsilon-2\alpha\epsilon(1+\ln(n)/r)}+\epsilon\ln(r). \tag{7}$$

Inequality (7) is satisfied if $3\epsilon-2\le\epsilon-2\alpha\epsilon-2\delta$ for any positive constant δ and this is satisfied if $\alpha\le(1-\epsilon-\delta)/\epsilon$. Thus, if $1>\epsilon>\frac{2}{3}$ and $pr\le(1-\epsilon-\delta)\ln(n)/\epsilon$, then (2) is polynomially bounded.

The remaining case, $pn\to 0$ and $pr<\gamma\ln\ln(n)$, is straightforward. □

We make four remarks about the proof of Theorem 3.1. First, in (6), only the term $e^{-2r^\alpha-2r^{2\alpha-1}}r^\epsilon$ is due to the presence of null clauses in I. But this term is ignored when determining that $\alpha\le\frac{2}{3}-\epsilon$ in the sentence following (6). Thus, the polynomial average time result for $n=r^\epsilon$, $\epsilon<\frac{2}{3}$, is *not* due to the presence of null clauses in I.

Second, in (7) the term $r^{\epsilon-2\alpha\epsilon(1+\ln(n)/r)}$ is due to the presence of null clauses and the term $r^{3\epsilon-2}$ is due to the removal of infrequent variables. Both the null clause check and removal of infrequent variables account for polynomial average time when $pr<(1-\epsilon)\ln(n)/\epsilon$, $\frac{2}{3}<\epsilon<1$. That is, neither checking for null clauses alone nor removing infrequent variables without checking for null clauses is powerful enough to achieve this result.

Third, if $pr<(\ln(n)/4)^{1/3}r^{2/3-\epsilon}$, $0<\epsilon<\frac{2}{3}$, then the average number of serious variables, although possibly an increasing function of n, is small compared to the number of infrequent variables. Thus, in this case it can be said that *INFREQ* works well because nearly all the variables are eliminated by resolution, the pure literal rule, unit elimination, double elimination, and trivial elimination. On the other hand, if $pr<(1-\epsilon)\ln(n)/\epsilon$ and $\frac{2}{3}<\epsilon<1$, then it can turn out that many variables are serious. In fact, if $pr=(1-\epsilon)\ln(n)/\epsilon$, $\frac{2}{3}<\epsilon<1$, then the average number of serious variables is $\theta(\ln^3(r)r^{3\epsilon-2})$. Exhaustive search over so many variables would require superpolynomial time. However, *INFREQ* works well on the average in this case too.

Removing resolution (recall this is on double variables only) and the null clause check from *INFREQ* leaves essentially the algorithm that was analyzed in [1] called *PLR*. But *PLR* requires superpolynomial average time if $pr<(1-\epsilon)\ln(n)/\epsilon$, $\frac{2}{3}<\epsilon<1$. Thus, resolution and the null check account for the good average performance of *INFREQ* in this case. Moreover, *PLR* requires superpolynomial average time even if $pr>\sqrt{\ln(n)}\cdot r^{1/2-\epsilon}$, $\frac{1}{2}<\epsilon<\frac{2}{3}$. Thus, in the case where $\sqrt{\ln(n)}\cdot r^{1/2-\epsilon}<pr<(\ln(n)/4)^{1/3}r^{2/3-\epsilon}$, $\frac{1}{2}<\epsilon<\frac{2}{3}$, only the addition of resolution on double variables to

PLR accounts for polynomial average time. This means that with $\frac{1}{2} < \epsilon < \frac{2}{3}$, large samples of instances with up to r^{α} literals per clause on the average, $0 < \alpha < \frac{1}{6}$, can be solved in polynomial average time with *INFREQ*, whereas with *PLR*, superpolynomial average time is required even if the average number of literals per clause is vanishingly small. It is perhaps surprising that such a small change to *PLR* can have such an effect on average case performance.

The fourth remark concerns the scope of infrequent variables. From the paragraph preceding Theorem 3.1, it should be evident that, if $pn \to 0$ and only unit variables are eliminated in *INFREQ*, then $\mu = \theta((np)^2 r)$ and, up to constant factors, (6) becomes

$$\ln(n) r^{2\alpha+2\epsilon-1} \le r^{\epsilon} e^{-2r^{\alpha}-2r^{2\alpha-1}} + \epsilon \ln(r),$$

which is satisfied if $\alpha < \frac{1}{2} - \epsilon$. If we could use substitution rules to eliminate triple variables, those which appear three times in an instance, then $\mu = \theta((np)^4 r)$ and $\alpha < (\frac{3}{4}) - \epsilon$. If we could eliminate all variables occurring i or fewer times in I, then we would have $\mu = \theta((np)^{(i-1)} r)$ and polynomial average time if $pr < r^{i/(i+1)-\epsilon}$, $\epsilon < i/(i+1)$. Clearly, i does not have to be very large to make a major impact on the parameter space supporting polynomial average time. Unfortunately, trying to eliminate even triple variables can cause an exponential explosion of the size of I. In this event the assumption that the complexity of each step of *INFREQ* is polynomially bounded is not valid. We ask: are explosions so infrequent that they do not significantly affect average time performance? An affirmative answer would have a major impact on polynomial average time results under the random-clause-size model. We leave investigation of this question for a future paper.

The next theorem shows where *INFREQ* runs in polynomial average time when $n = \beta r$, β a positive constant.

THEOREM 3.2. *INFREQ runs in polynomial average time if* $n/r = \beta$, *where* β *is a positive constant, and* $4.15(1-(1-p)^{2\beta r}(1+2\beta pr+2(\beta pr)^2)) < \beta e^{-2pr}$.

Proof. Since $p < 1$, $1/(1-p) > 1$, and $1/(1-p)^2 > 1$, then

$$\mu = (1-(1-p)^{2n}(1+2pn/(1-p)+2(pn)^2/(1-p)^2))r \le (1-(1-p)^{2n}(1+2pn+2(pn)^2))r.$$

Thus, (3) is polynomially bounded if

$$\begin{aligned} & -ne^{-2pr} + ln(2)(6(1-(1-p)^{2n}(1+2pn+2(pn)^2)))r \le \ln(n) \iff \\ (8) \qquad & -\beta e^{-2pr} + 4.15(1-(1-p)^{2\beta r}(1+2\beta pr+2(\beta pr)^2)) \le \ln(n)/r. \end{aligned}$$

The theorem follows. □

According to Theorem 3.2, *INFREQ* has polynomial average time if $2pr < \ln(\beta) - \ln(4.15)$ (this is fairly tight if β is large). If $\beta = 1$, then *INFREQ* has polynomial average time if $pr < .5$.

4. Conclusions. We have investigated a simple strategy for solving instances of CNF Satisfiability with respect to average case performance. The important idea is the elimination of infrequent variables before applying, in this case, exhaustive search. We have shown that this strategy is superior in average case performance to all other algorithms analyzed under the random-clause-size model when $pr < \sqrt{\epsilon r \ln(r)}$, $n < r^{\epsilon}$, and $\epsilon < 1$. The strategy may be generalizable, to some extent, and the analysis seems to suggest the outcome of an investigation of such a generalization.

REFERENCES

[1] K. Bugrara, Y. Pan, and P. W. Purdom, *Exponential average time for the pure literal rule*, SIAM J. Comput., 18 (1989), pp. 409–418.

[2] J. Franco, *On the probabilistic performance of algorithms for the Satisfiability problem*, Inform. Process. Lett., 23 (1986), pp. 103–106.

[3] K. Iwama, CNF *Satisfiability test by counting and polynomial average time*, SIAM J. Comput., 18 (1989), pp. 385-391.

[4] P. W. Purdom and C. A. Brown, *The pure literal rule and polynomial average time*, SIAM J. Comput., 14 (1985), pp. 943–953.

[5] ———, *Polynomial average time Satisfiability problems*, Inform. Sci., 41 (1987), pp. 23–42.

[6] ———, *The Analysis of Algorithms*, Holt, Rinehart Winston, New York, 1985.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1128–1147, December 1991
© 1991 Society for Industrial and Applied Mathematics
009

PARALLEL TREE CONTRACTION PART 2: FURTHER APPLICATIONS*

GARY L. MILLER[†] AND JOHN H. REIF[‡]

Abstract. This paper applies the parallel tree contraction techniques developed in Miller and Reif's paper [*Randomness and Computation, Vol.* 5, S. Micali, ed., JAI Press, 1989, pp. 47–72] to a number of fundamental graph problems. The paper presents an $O(\log n)$ time and $n/\log n$ processor, a 0-sided randomized algorithm for testing the isomorphism of trees, and an $O(\log n)$ time, n-processor algorithm for maximal subtree isomorphism and for common subexpression elimination. An $O(\log n)$ time, n-processor algorithm for computing the canonical forms of trees and subtrees is given. An $O\log n$ time algorithm for computing the tree of 3-connected components of a graph, an $O(\log^2 n)$ time algorithm for computing an explicit planar embedding of a planar graph, and an $O(\log^3 n)$ time algorithm for computing a canonical form for a planar graph are also given. All these latter algorithms use only $n^{O(1)}$ processors on a Parallel Random Access Machine (PRAM) model with concurrent writes and concurrent reads.

Key words. parallel algorithms, tree contraction, graph isomorphism, graph connectivity, subexpression, elimination

AMS(MOS) subject classifications. 05C05, 05C10, 05C40, 68Q25, 68R25

1. Introduction. In the previous companion paper [29], we introduced a bottom-up technique for processing a tree which we named *Parallel Tree Contraction.* This technique is in many cases preferable to previously utilized top-down techniques for processing trees which require a precomputation to find the nodes of a tree that separated the tree into pieces of size at most $\frac{2}{3}$ the tree size. Our first paper considered expression evaluation as our first example and prime application of CONTRACTION. Our main results were an $O(\log n)$ time using n processor deterministic algorithm, as well as an $O(\log n)$ time using $n/\log n$ processor randomized algorithm for tree contraction. The example and application of tree contraction given in Part I [30] were dynamic expression evaluation. Part II will give some further applications. This second paper presumes that the reader has knowledge of our companion paper. The goal of this paper is to apply CONTRACTION to a wide variety of graph problems.

We will assume throughout this paper the Parallel Random Access Machine Model (PRAM), which we also assume can perform concurrent reads and writes (see [40]).

The discussion begins in §2, where we present a zero-sided randomized algorithm which tests the isomorphism of trees in $O(\log n)$ time using $n/\log n$ processors, and which tests the isomorphism of maximal subtrees and subexpressions in $O(\log n)$ time using n processors. We also exhibited a deterministic $O(\log n)$ time algorithm which uses $n \log n$ processors for computing the canonical forms of trees. Previously, Ruzzo [33] showed that isomorphism of trees of degree at most $\log n$ could be tested in $O(\log n)$ time. No polylogarithmic parallel algorithm was previously known for isomorphism of unbounded-degree trees.

* Received by the editors August 31, 1987; accepted for publication (in revised form) December 20, 1990. The preliminary version of this paper appeared in 26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, 1985, pp. 478–489.

† School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890. The work of this author was supported in part by National Science Foundation grant CCR-8713489.

‡ Computer Science Department, Duke University, Durham, North Carolina 27706. The work of this author was supported in part by DARPA/ARO contract DAAL03-88-K-0195, Air Force contract AFOSR-87-0386, DARPA/ISTO N00014-80-C-0458 and N00014-91-J-1985, NASA subcontract 550-63 of primecontract NAS5-30428.

In §3, the tree of 3-connected components (as defined by Hopcroft and Tarjan [16]) is constructed in $O(\log n)$ time on a PRAM. Previously, Ja'Ja' and Simon [18] gave an $O(\log n)$ time PRAM algorithm for finding maximal subsets of vertices, which are pairwise 3-connected; but they did not address the problem of finding the tree of 3-connected components. In the case of 3-connected graphs, they constructed the planar embedding in $O(\log^2 n)$ time on an Exclusive Read and Exclusive Write (EREW) PRAM, but it is easy to see that their algorithm required only $O(\log n)$ time, using the Concurrent Read and Concurrent Write (CRCW) model. They did not construct embeddings of general planar graphs. In §3, an $O(\log^2 n)$ time PRAM algorithm is given that computes the explicit planar embedding of planar graphs even if the graphs are not 3-connected.

Section 4 presents an $O(\log^3 n)$ time PRAM algorithm that computes a canonical form for planar graphs. No polylogarithmic parallel algorithm for testing the isomorphism of planar graphs previously existed.

Section 5 presents an NC reduction from the problem of computing canonical forms of a general graph to the problem of canonical forms for 3-connected graphs. This is an $O(\log n)$ time reduction using $n^{O(1)}$ processors on a PRAM.

Finally, §6 references extension and further applications of the parallel tree contraction technique that have been done since the original writing of this paper.

All our PRAM algorithms use only a polynomial number of processors. Effort shall be taken to minimize the number of processors used. Most of these results can also be expressed in terms of circuits with simultaneous depth: $(\log n)^c$ and n^k size, for fixed constants c and k.

2. Isomorphism and canonical labels for trees. Let T and T' be two rooted trees with roots r and r' and vertex sets $V(T)$ and $V(T')$, respectively, where $|V(T)| = n$. T is *isomorphic* to T' if there exists a bijective map from $V(T)$ to $V(T')$ which preserves the parent relation. A map L from trees to strings such that T is isomorphic to T' if and only if $L(T) = L(T')$ is called a *canonical label.* A subtree T' of a rooted tree T is said to be an *induced subtree* if there exists a vertex v of T such that the vertices of T' are v and all the descendants of v in T. This paper considers only the induced subtrees. Thus, a subtree is assumed to be an induced subtree (note induced subtrees are also termed maximal subtrees in the literature). *Canonical labels for all induced subtrees* of a tree T is a map L from $V(T)$ to finite strings such that for all $x, x' \in T$ the subtree rooted at x is isomorphic to the subtree rooted at x' if and only if $L(x) = L(x')$. All results to follow will apply to unrooted trees as well.

Canonical labels for all induced subtrees can be used for code optimization. Here, one merges all nodes with common labels producing an acyclic digraph. This process is called *common subexpression elimination.* First, a randomized algorithm for tree isomorphism is presented.

The *height* $h(v)$ of a node v in a tree T is the maximum distance from v to any of its leaves. That is, $h(v) = 0$ if v is a leaf; otherwise, if v has children, $v_1, \cdots, v_k$, then $h(v) = 1 + \max\{h(v_i) | 1 \leq i \leq k\}$. It is a straightforward exercise to see that the height of all nodes in a tree with n nodes can be computed deterministically by *Parallel Tree Contraction* (PTC) in $O(\log n)$ time using n processors, or alternatively, by using $n/\log n$ processors by the randomized version of *Parallel Tree Contraction*, as discussed in the first part of this paper [29].

A multivariate polynomial Q_v is canonically associated with each vertex v of the tree T. Let $x_1, x_2, \cdots$ be distinct independent variables. For each leaf v, set $Q_v = 1$. For each internal node v of height h with children $v_1, \cdots, v_k$, set $Q_v = \prod_{i=1}^{k}(x_h - Q_{v_i})$

using induction on the height h. Thus, the polynomial Q_r of the root r with height h is a polynomial $Q_T(x_1, \cdots, x_h)$ of degree less than or equal to n. Q_r is viewed as a polynomial over a finite field F. Using the fact that polynomial factorization is unique over F, Lemma 2.1 follows.

LEMMA 2.1. *The subtrees rooted at v and v' are isomorphic if and only if $Q_v = Q_{v'}$ over a field F.*

To test if a polynomial $Q(x_1, \cdots, x_h)$ of degree less than or equal to n is identically zero, an old idea, which goes back at least as far as Edmonds, is used [34]. The polynomial is evaluated at a random point and checked to see if the value is nonzero. In this section the following technical lemma is used which is similar to a lemma in [17].

LEMMA 2.2. *If F is a finite field of size p, p prime, such that $p \geq n^{\alpha+1}h$, $\alpha \geq 1$, $\vec{a}$ is a random element of F^h, and $Q(x_1, \cdots, x_h)$ is a polynomial of degree less than or equal to n which is not identically zero over F, then* $\text{Prob}[Q(\vec{a}) = 0] \leq 1/n^\alpha$.

Proof. We first show by induction on h (see [17]) that the polynomial Q has at least $(p - n)^h$ points for which it is not zero. For the case $h = 1$, Q has at most n roots out of a possible p elements. Thus, Q has at least $p - n$ nonzero points. Suppose the claim is true for all polynomials with h variables, and Q is polynomial in at most $h + 1$ variables. In this case, Q can be written as a polynomial in the first variable x with coefficients being polynomials in at most h variables. At least one of the coefficients Q_1 must be a polynomial which is not identically zero. Thus, there are at least $(p-n)^h$ points for which Q_1 is not zero. Now, for each one of these points there are $p - n$ values of x in F for which Q is not zero. Therefore, Q has at least $(p - n)^{h+1}$ points for which it is not zero. Since $\vec{a}$ is a random element of F^h, the above can be written as a probability: $\text{Prob}[Q(\vec{a} \neq 0)] \geq (p - n)^h/p^h = (1 - n/p)^h$.

Substituting $n^{\alpha+1}h \leq p$ for p yields $\text{Prob}[Q(\vec{a}) \neq 0] \geq (1 - (1/n^\alpha h))^h$. Since $(1 - (1/n^\alpha h))^h \geq (1 - 1/n^\alpha)$, the desired inequality, $\text{Prob}[Q(\vec{a}) = 0] \leq 1/n^\alpha$, is obtained. □

The tree isomorphism algorithm is described in procedure form (see Fig. 1). Two different procedures have actually been given, depending on whether one implements step (1) or step (1′). If step (1′) is implemented, it must have access to a very small table of at most $O(\log n)$ prime integers. This table of prime integers, PT, needs to only contain one prime between 2^t and 2^{t+1} for each t. The existence of the primes is guaranteed by Bertrand's postulate (see [15]). As Theorem 2.3 will show, isomorphism of trees of size less than or equal to n can be tested using a table of $O(\log n)$ primes, each of value less than or equal to $n^{O(1)}$. This table can be generated in random polynomial time. To generate the table of primes, we need an estimate on the number of primes in an interval of size n to $2n$ (see [32] and a random polynomial-time primality test, [35], [24]). However, if only step (1) is used, a uniform algorithm in the usual sense is obtained. Our analysis of the uniform algorithm shows only that the probability of error is less than $\frac{1}{2}$. On the other hand, the probability of error using the table of primes is at most $1/n$. In step (4), the *Asynchronous Tree Contraction* algorithm [29] is used, since the time to RAKE a node with k children will be $O(\log k)$.

THEOREM 2.3. *Randomized$_1$ Tree Isomorphism using step (1) tests tree nonisomorphism in $O(\log n)$ time using $n/\log n$ processors with the probability of error less than or equal to $1/2$. If a table of primes is given, then the procedure works with a probability of error of at most $1/n^\alpha$.*

Proof. The case when a table PT of primes is used follows by a straightforward

Procedure Randomized$_1$ Tree Isomorphism (One-Sided)

(1) Pick a random integer m in the range $(hn^{\alpha+1})^2 \leq m \leq 2(hn^{\alpha+1})^2$.
(1′) Pick a prime m in the range $hn^{\alpha+1} \leq m \leq 2hn^{\alpha+1}$ of the given list of primes PT.
(2) For each node v of T or T', assign the polynomial Q_v to v as described above.
(3) Assign to each x_i a random value between 1 and m.
(4) Evaluate Q_T and $Q_{T'}$ using one of our dynamic expression evaluation algorithms [29] and return w and w', respectively.
(5) **If** $w \neq w'$, **then** output "not isomorphic," **else** output "isomorphic."

FIG. 1. *A one-sided randomized tree isomorphism test.*

calculation using the last lemma. In this case, the algorithm tests if the polynomial $Q = Q_T - Q_{T'}$ is identically zero or not. By the last lemma, the probability that a random element is a zero of Q is at most $1/n^\alpha$.

Suppose a random integer is used instead of picking a prime from a table. In this case, the probability that the largest prime factor of a random integer m has size at least $\sqrt{m}$ is at least $\frac{2}{3}$ (see [20]). For Q to be zero at some point modulo m, it must be zero modulo p. Thus, at least $\frac{2}{3}$ of the time, m will have a prime factor of size at least $hn^{\alpha+1}$, in which case steps (2)–(5) will be executed with an error of at most $1/n^\alpha$. For a sufficiently large n the probability of error is at most $\frac{1}{2}$. □

Note that the main source of error is step (1), not steps (2)–(5). This fact is used in the next algorithm. Next, the algorithm is modified into a zero-sided randomized algorithm, i.e., one that never makes an error. The idea of the algorithm will be to modify *Procedure Randomized$_1$ Tree Isomorphism* so that it outputs a value for each subtree of T and T'. Assuming that these values are the correct labels for each subtree, these values are used to find an isomorphism. Note that we can easily test whether or not this map is an isomorphism. This modified procedure is called *Randomized$_1$ Label Generation.* More precisely, steps (4) and (5) are replaced with a step that evaluates all subpolynomials.

This new algorithm will also canonically label the set of all induced subtrees of a tree. But this does not give a canonical label for trees, since there is an exponential number of trees and only a polynomial number of labels. This last problem will be addressed later on in the paper.

The problem of testing the isomorphism of trees can be reduced to the problem of canonically labeling all induced subtrees of a tree, as follows:

- Viewing the two trees as subtrees of a larger tree.
- Asking for the labeling of all its subtrees.
- Checking whether or not the labels on the two roots of the subtrees are the same.

Thus, our attention is restricted to the problem of canonically labeling all induced subtrees. The following lemma will be used here.

LEMMA 2.4. *A map L is a canonical labeling of all induced subtrees of T if and only if:*

1. *If v, v' are leaves, then $L(v) = L(v')$;*
2. *$L(v) = L(v')$ if and only if $\{L(v_1), \cdots, L(v_k)\} = \{L(v'_1), \cdots, L(v'_k)\}$, where $v_1, \cdots, v_k$ are the children of v and $v'_1, \cdots, v'_k$ are the children of v'.*

Proof. The proof is a straightforward induction on the height of subtrees. One must show that two subtrees are isomorphic if and only if they have the same labels. Condition 1 states that subtrees of height 0 (leaves) are isomorphic, while condition 2 gives us the inductive step. □

The labels generated by *Procedure Randomized*$_1$ *Label Generation* clearly satisfy condition 1. Only condition 2 remains. Note that if $\{L(v_1), \cdots, L(v_k)\} = \{L(v'_1), \cdots, L(v'_k)\}$, then, clearly, $L(v) = L(v')$. Thus, one tests only that nodes with the same label have the same set of labels on their children. One simply sorts the nonleaf vertices by their label value obtaining ordered linked lists of vertices with the same labels. It will suffice to check that consecutive vertices with the same label have children that have the same set of labels. To test this latter condition for each node, one must sort the labels of each node's children. Next, only pairs of linked lists are checked for equality. Thus, all subtrees can be canonically labeled in the cost of two sorts of less than or equal to n numbers where each number is of the size $O(\log n)$. Both randomized and deterministic algorithms using $O(\log n)$ time and n processors are known for sorting [2], [31], [8].

Using this result yields the following theorem.

THEOREM 2.5. *Tree isomorphism and common subexpression elimination can be performed with a 0-sided randomized algorithm in $O(\log n)$ time using n processors with an error probability of $1/n$, given a table of $O(\log n)$ primes each of value less than or equal to $n^{O(1)}$; otherwise, the error probability is at most $\frac{1}{2}$.*

Proof. The tree T to be labeled will have n associated polynomials, one for each subtree. *Procedure Randomized*$_1$ *Label Generation* must be run with enough reliability so that any two of the n polynomials will have distinct values if their subtrees are not isomorphic. In the worst case, the difference of all pairs of polynomials must have a nonzero value. This implies that $\alpha = 3$ can be picked so that the probability of any one of the n^2 polynomials being nonzero will be at most $1/n^3$. In the case where a random integer is picked; i.e., step (1) is executed, simply note that the probability of error is at most $\frac{1}{3}$ and it comes only for the first step, not the others. Thus, the random integer case works with a probability of error of at most $\frac{1}{2}$. □

The remainder of this section exhibits a fast deterministic algorithm for canonical labelings of trees. Note that the randomized procedure developed in Theorem 2.5 does not produce canonical forms for trees. Canonical forms can be obtained by using sorting. The idea is to assign canonical labels to the nodes inductively by height. The leaves are labeled with zero. Suppose, inductively, that the children, $v_1, \cdots, v_k$, of v have labels $L(v_1), \cdots, L(v_k)$; then the label of v will be the concatenation of the sorted list of labels $L(v_1), \cdots, L(v_k)$, including a left and right parenthesis. By Lemma 2.4 this gives a canonical label for trees. This definition of the label for T seems hard to implement in parallel since a label which takes a long time to compute may have a small lexicographic value. This problem is solved by first sorting the children of a node based on the time in which its label was computed and then sorting the children on their label value.

The discussion begins with a simpler $O(\log^2 n)$ time parallel algorithm. Here, the children of a node are sorted when all but at most one child has its label. If this final child exists, it is placed at the end of the list. A fixed place in the list is left for the missing value. A node at an intermediate point of the algorithm which has one child may be viewed as having a label with one free variable. The intended value of the variable is the label of the child. Thus, if its child also has only one child and its label

has been computed up to a free variable, then the labels may be composed; i.e., apply COMPRESS.

Since the labels may be as large as $O(n)$ long, it is unreasonable to expect that two labels can be compared by one processor in unit time. However, two characters can certainly be compared in $O(1)$ time by one processor. This implies the following well-known lemma.

LEMMA 2.6. *The comparison of two strings of length n can be performed in $O(1)$ time using n processors.*

Theorem 2.7 follows from the preceding lemma.

THEOREM 2.7. *Canonical labelings for trees can be computed in $O(\log^2 n)$ time using n processors.*

To see that the above algorithm works in $O(\log^2 n)$ time, simply note that each RAKE takes at most $O(\log n)$ time and that CONTRACT is applied at most $O(\log n)$ times by the results of [29]. The bound of n on the number of processors is obtained as follows. Initially only the leaves have their labels, and the sum of their lengths is at most n. The labels on internal nodes will be the concatenation of the leaf labels below it plus separating symbols, say, left and right parentheses. Thus, the length of the label of an internal node is linear in the number of nodes in its subtree. Since only leaves are ever sorted by the algorithm, the sum of the length of the strings sorted in any RAKE is at most $O(n)$. Thus, we need only n processors. □

Our $O(\log n)$ time algorithm is slightly more complicated. Our approach begins by sorting labels at a node as soon as they arrive. That is, we first order the children of a node based on the time each child's label arrives. Among those children whose labels arrived at the same time, we further order them by their label values. In general, this labeling returns a different canonical form and label from the previous algorithms, but it is also canonical, since the ordering of the tree is, up to isomorphism, independent of how the tree is given.

Ignoring for the moment the cost of collecting labels together so that they may be sorted in parallel, the algorithm will take $O(\log k)$ steps to remove the k leaves of a node. Thus we have an algorithm which removes the k leaves of a node in $O(\log k)$ and, therefore, by the results of [29], it will run for only $O(\log n)$ time when run asynchronously.

The labels that arrive at the same time must be coalesced so that they are "ready" to be sorted. We cannot afford to coalesce the labels after they arrive, since the cost to coalesce the labeled children may be a function of all the children of the node; thus, the overall running time may grow faster than $O(\log n)$. We circumvent the problem of coalescing the labels on-line by simply computing when the labels will arrive, without sorting, followed by a second phase where we sort these "times" offline.

Recall that each nonleaf node v has associated with it an array of storage locations, one for each child. Each storage location is used for the label of the child and will be used when its label has been computed. In the preprocessing phase, the storage locations are rearranged by sorting the children by arrival times.

As mentioned above, the time when a given value will arrive in the preprocessing phase is determined without actually computing the values. These times are then used to sort the children of each node. Let c be an integer greater than or equal to 4, such that deterministic parallel sorting of $k \geq 2$ numbers can be performed in $f(k) = c\lceil \log k \rceil + 2 + \delta$ time on a Concurrent Read and Concurrent Write (CRCW) PRAM, where δ is a constant yet to be determined. Since $f(k)$ can be easily computed, the parallel sorting algorithm can be slowed down so that it takes exactly $f(k)$ time

to sort a string of length k. Let the *label-time* of a node in a tree T be the time at which the node gets its label when the hypothetical canonical labeling algorithm is run on T. Next, the label-time for each node is computed.

Both RAKE and COMPRESS of the above algorithm assume the labels that need to be sorted are consecutive. COMPRESS is a straightforward simulation, since each COMPRESS step takes only unit time. The simulation of RAKE is more subtle. We will now show how to determine when each node becomes either a leaf or a parent of a single child. The label-time of a leaf is 1. If a node v is at no time the parent of a single child, then the label-time of v is $\max\{f(K_i) + i\}$, where K_i is the number of children of v whose label-time is i. If, at some point, v becomes the parent of a single child, then that time will be $\max\{f(K_i) + i\}$, where the maximum is over all children except for the last child processed. Then label-time can be computed by the simulation of COMPRESS. In either case, only the value $\max\{f(K_i) + i\}$ need be computed on or before time $\max\{f(K_i) + i\}$. The value is actually computed by time $\max\{2\lceil\log K_i\rceil + i + 4\}$ (see Lemma 2.8). First, the K_i's are computed, then the $\max\{f(K_i) + i\}$ is computed from the K_i's in unit time.

By the results from [29], the largest value of any label-time will be at most $O(\log n)$. A vector of integers is initially associated with each storage location of a nonleaf node v, and all entries are zero. If the label-time for the child of a node arrives at time i, then 1 is added to position i of this vector, and the vector is marked to indicate that its time is known. A marked vector can be combined with a neighboring left or right vector, either marked or unmarked. The combination of the two vectors is simply the vector sum, and this procedure is considered a COMPRESS-like operation applied to consecutive vectors. If only one of the two vectors is marked, then the combined vector is considered unmarked; otherwise, it is considered marked. We assume that we have $O(\log n)$ processors per node.

We shall implement the above compress-like operation using a variant of Wyllie's algorithm for list-ranking [40]. We consider our list of vectors as a linked-list. As in Wyllie's algorithm, the last element points to nil. For booking reasons, add a new pointer at the beginning of the list. The algorithm finishes when the new beginning pointer points to nil. At each stage, a node may update its pointer if it is pointing to a marked vertex that is not nil. When a node updates its pointer, it also adds the value of the parent's vector to itself. Therefore, this is a CREW algorithm.

A maximal consecutive sequence of marked vectors is called a *run*. Note that the above procedure applied to a *run* will decrease the length of the run by at least $\frac{1}{2}$. At some point, the sequence of vectors will be reduced to a single vector (the new vector added to the beginning of the list) whose ith value is K_i. In unit time, K_i is replaced with $f(K_i) + i$. Also, in unit time, the maximum of $\log n$ values can be computed using $O(\log^2 n)$ processors. We use a processor P for each pair of values. The processor P will cancel the smaller of its two values. The remaining value is the maximum. We will assume that the above two-unit time calculations are performed in at most δ machine steps.

It remains to show that the vector values K_i are computed "on time." That is, the vector of values K_i is computed by time $\max\{f(K_i)+i-\delta\} \le \max\{4\lceil\log K_i\rceil+i+4\}$ for each node. The problem is abstracted to the following conceptually easier problem: a list of characters, each of which is initially the letter I for inactive, is presented; i.e., the string I^n is given. At time i, a subset of K_i of the characters I change, to A. Each A is now thought of as an active character. At each time step, a run of t A's is replaced by a run of $\lfloor t/2\rfloor$ A's. This process is called *ACTIVATE and COMPRESS.*

LEMMA 2.8. *The process ACTIVATE and COMPRESS will terminate in the empty string by a time of at most* $\max\{2\log K_i + i + 2\}$.

Proof. Suppose that $K_1, \cdots, K_m$ is a sequence of activations where m equals the maximum i such that $K_i \neq 0$. Further, let $l = \max\{2\log K_i + i\}$, for $i = 1$ to m. Note that $l \geq m \geq 1$.

Let Δ_i be the number of A's in the string at time i after the ith list activation. At time i, there are K_i A's added to the string while COMPRESS reduces the number of A's by one-half. Thus, the contribution of the K_i As at time $t \geq i$ is bounded by $K_i/2^{t-i}$. This gives the following inequality for $t \geq m$:

$$\Delta_t \leq \frac{K_1}{2^{t-1}} + \cdots + \frac{K_m}{2^{t-m}}. \tag{1}$$

Using the fact that for all i, $2\log K_i + i \leq l$ implies $K_i \leq 2^{(l-i)/2}$, we substitute this inequality into (1), yielding

$$\Delta_l \leq \frac{1}{2^{l/2}} + \cdots + \frac{1}{2^{(l-m)/2}}.$$

Since the right-hand side is a geometric series in $1/\sqrt{2}$ beginning with $1/\sqrt{2}$, it follows that $\Delta_l \leq 1/(\sqrt{2}-1) < 3$. Since Δ decreases by at least $\frac{1}{2}$ at each time step, and it is integral, we get $\Delta_{l+2} = 0$. Therefore, $l + 2 = \max\{2\log K_i + i + 2\}$; this proves the lemma. □

THEOREM 2.9. *Canonical labelings for trees can be computed in* $O(\log n)$ *time, using* $O(n\log n)$ *processors.*

Proof. The algorithm consists of three majors steps, as summarized below:

1. Compute the label-time of each vertex.
2. Sort and order the children of each node up to their label-time value.
3. Compute the final ordering of the children by computing vertex labels using sorting.

Using Lemma 2.8, the label-time values for each node can be computed on or before its label-time. The label-time of a node is not passed to its parent until the actual time of the label-time value, thus preserving the invariant property that label-time values arrive at the actual time of the label-time value. Therefore, step 1 takes $O(\log n)$ time using $O(\log n)$ processors per node ($n\log n$ in total).

In step 2, the children can be sorted at a node by their label-time values in $O(\log n)$ time using n processors. Finally, in step 3, the labels can be computed by sorting label values. As in Theorem 2.3, the timing analysis of Theorem 6.1 from [29] can be applied to give an $O(\log n)$ time bound. Again, using the analysis from the proof of Theorem 2.3 to step 2 of procedure Randomized$_1$ Tree Isomorphism, this algorithm requires at most n processors to achieve the $O(\log n)$ time bound. □

This motivates another generalization of *Parallel Tree Contraction* which will be used to compute the 3-connected components of a graph in $O(\log n)$ time, instead of $O(\log^2 n)$ time.

Consider *Asynchronous Parallel Tree Contraction*, as defined in Part 1 [29], applied to an ordered tree of unbounded degree, where the RAKE operation is restricted to removing a constant proportion of consecutive leaves. In particular, assume that RAKE replaces a run of length k by a run of length $\lfloor k/2 \rfloor$ in unit time. Thus, COMPRESS acts on chains, and RAKE acts on runs. Recall from Part I that a *chain* in a rooted tree is a sequence of vertices $v_1, \cdots, v_t$ such that v_{i+1} is the only child of v_i,

for $1 \leq i < t$. If the tree is undirected, a *chain* will be a sequence of vertices $v_1, \cdots, v_t$ such that v_{i-1} and v_{i+1} are the only neighbors of v_i, for $1 < i < t$. It is crucial that a vertex be processed under COMPRESS when it has one child that is not a leaf, or possibly two children that are leaves, a leftmost and, possibly, a rightmost child; i.e., one or two runs of length 1. This procedure is called *Parallel Tree Contraction with RAKE restricted to runs.*

THEOREM 2.10. *Parallel Tree Contraction with RAKE restricted to runs requires only* $O(\log n)$ *applications to reduce a tree to a single vertex.*

3. Computing the 3-connected components. The main goal of this section is to give a new parallel algorithm for decomposing a graph into a tree of 3-connected components. To this end, we first discuss the decomposition of a graph into a tree of 2-connected components. We then discuss prior work on the decomposition of general graphs into their tree of 3-connected components, including a definition of brides and Hopcroft and Tarjan's use of virtual edges. Finally, we give our definition of the 3-connected components of a graph, and relate how to use *Parallel Tree Contraction* to find these components.

Two vertices v and w in an undirected graph $G = (V, E)$ are *k-connected* if there exist k paths in G from v to w which are pairwise vertex disjoint, except at their endpoints v and w. Thus, two vertices sharing k-edges are k-connected. The graph G is k-connected if every pair of vertices is k-connected.

Before giving our algorithm, which decomposes a connected graph into its tree of 3-connected components, we will discuss the decomposition of a connected graph into its tree of 2-connected components. This decomposition consists of three types of components. First, there are the proper 2-connected components. These are the subgraphs induced by a maximal subset of vertices which are pairwise 2-connected. Second, there are the articulation vertices or separating vertices. Finally, there are separating edges. The vertices of the tree consisting of 2-connected components are the components described above. An articulation vertex is adjacent to another component if it is contained in the component. Recently, Tarjan and Vishkin [36] have shown how to construct the 2-connected components of a graph in $O(\log n)$ time using a linear number of processors on a PRAM. These components form a tree where a component and a separating vertex are adjacent if the vertex is contained in the component. However, the 3-connected components are more difficult to define and seem to require a more sophisticated algorithm.

Hopcroft and Tarjan [16] give a precise algorithmic definition, which will be reviewed below, of the 3-connected components and show how any graph can be decomposed uniquely into a tree of 3-connected components. In the same paper, they also give a linear time algorithm for finding the tree of 3-connected components. Unfortunately, it is a highly sequential algorithm. A related distinct question is finding the maximal subsets of vertices of size greater than or equal to 2 which are pairwise 3-connected. These subsets shall be called the *3-sets* of G. Ja'Ja' and Simon [18] give an algorithm using $O(\log n)$ time and $n^{O(1)}$ processors for finding these 3-sets. There is a unique 3-connected graph associated with each 3-set. The proof and construction can be obtained by Lemma 3.1.

First, we will define the notion of a bridge. Let $C \subset V$. Two edges e and e' of G are *C-equivalent* if there exists a path from e to e' avoiding C. The induced graphs on the equivalence classes of the C-equivalent edges are called the *bridges* of C. A bridge is *trivial* if it consists of a single edge. A pair of vertices is a *separating pair* if it has 3 or more bridges or 2 or more nontrivial bridges. A *3-connected separating*

pair is a pair of vertices which is both 3-connected and a separating pair.

LEMMA 3.1. *If $C \subset V$ is a 3-set of G, then each bridge of C contains at most 2 vertices in C. If G is 2-connected, then the bridge contains exactly 2 vertices of C.*

Proof. Suppose that some bridge B of C contains three vertices x_1, x_2, x_3 in C. Let p be a simple path from x_1 to x_3 in B. Let p_2 be a simple path from x_2 to a single vertex, y of p, such that $p_2 - y$ is disjoint from p. Let p_1, p_3 be the disjoint simple subpaths of p from y to x_1, x_3, respectively. Then p_1, p_2, p_3 are disjoint paths from y to distinct vertices x_1, x_2, x_3 of C. It follows that y is 3-connected to all the elements of C. This contradicts the assumption that C is a (maximal) 3-set. □

Throughout the discussion of the 3-connected components, we let G be the underlying graph, which is assumed 2-connected. The tree of 3-connected components consists of a tree of graphs called *components*. Two components are adjacent if they share an edge. These shared edges will not be edges from G, the original graph, but rather, from new edges called *virtual* edges. There will be exactly two copies of each virtual edge. Any vertex may appear in many components.

First, the graphs that will be the nodes in T will be described. The reader should be cautioned that, counter to intuition, the components are not always 3-connected graphs and separating pairs. The nodes of T are of three types: proper components, cycles, and m-bonds. The m-bonds lie between the components (proper components and cycles). They are precisely described below in Fig. 2, where the decomposition of a graph into components is shown. Note that the virtual edges are indicated by dotted lines.

- A *proper component* C is a simple 3-connected graph. C can be defined in terms of G as follows: the vertices of C consist of a 3-set S of size greater than or equal to 4 (*proper* 3-set). Two vertices of C share an edge in C if they shared one or more bridges in G. Note that C is simple; it has no multiple edges. An edge from x to y of C will be an original edge from G if x and y share exactly one trivial bridge; otherwise, the edge will be a virtual edge.
- A *cycle component* C is a simple cycle. C can be defined in terms of G as follows: the vertices of C are a maximal subset of the vertices S, such that the bridges of S in G form a simple cycle of size 3 or more, with possible pairs in S containing multiple bridges. As in the case of proper components, a unique trivial bridge e of S becomes an edge of C; otherwise, a virtual edge is formed.
- An *m-bond component* C is a graph on two vertices sharing two or more edges. C can be defined in terms of G as follows: x and y are the vertices of C if they are 3-connected and separating. There is one edge in C for each bridge of $\{x, y\}$ in G. If the bridge is trivial, the original edge in C can be used. Otherwise, a virtual edge is used. Note that 2-bonds have been introduced between two proper components or a proper component and a cycle component, which do not appear in the Hopcroft–Tarjan [16] definition.

We say a component is *associated* with another component if the two have a nonempty intersection.

We will now describe a parallel method for constructing the tree of components from the above three types of components. Our idea is to apply *Parallel Tree Contraction*; chains are not compressed, but, rather, every other component is removed from a chain. Since every other component on a path in the tree T is an m-bond, we can remove every other component on a chain by eliminating the *proper* and the *cycle* components. Thus, *proper* and *cycle* components associated with either zero, one, or

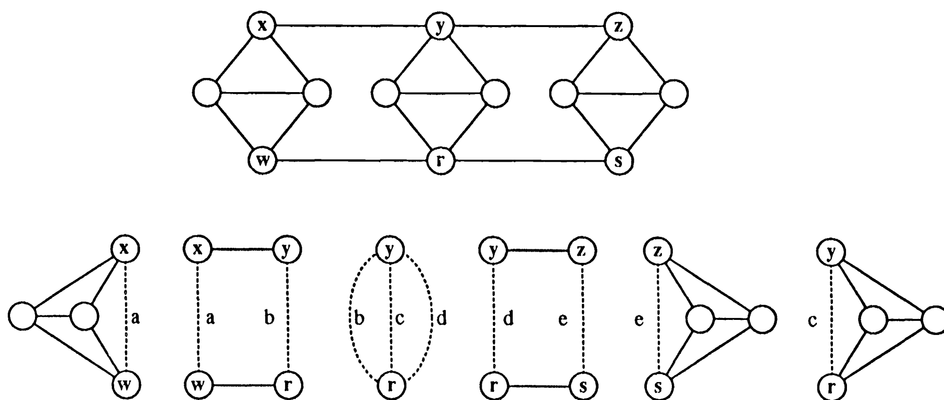


FIG. 2. *The decomposition of a graph into its 3-connected components.*

two other components are removed, as are m-bond components associated with either zero or one other component. All these components are removed in unit time except for the cycle components, which may take as much as $O(\log n)$ time; we will show how to amortize the cost in such a way that the total time decomposition is still only $O(\log n)$.

Using the work of Ja'Ja' and Simon [18] we compute the 3-sets and their bridges, along with the separating pairs and their bridges, in $O(\log n)$ time using $n^{O(1)}$ processors. Note that they also determine which separating pairs are 3-connected.

Assume that G is stored in memory as an incidence matrix and that the following information is maintained: a list of proper 3-sets; a list of 3-connected separating pairs; a forest indicating which 3-connected separating pairs are contained in which proper 3-sets; and, for each 3-connected separating pair $\{x, y\}$, a list of edges associated with it, partitioned according to which bridge of $\{x, y\}$ they belong. An edge e from x to y is *free* if x and y are not 3-connected. Note that the free edges will belong to the cycle components. A list of free edges is also maintained.

Let T be the tree of components of G. As components are removed from G, G will no longer be connected. Therefore, intuitively, G should be a collection of 2-connected graphs. But for technical reasons, the connected components of G may not be 2-connected. This complication will be discussed when the COMPRESS part of the algorithm is discussed.

The discussion will begin with RAKE. Here, one must determine when a component becomes a leaf in T, at which time it is removed. Note that a component is a leaf if and only if it contains zero or one nontrivial bridge. The case when a component has exactly one nontrivial bridge will be discussed first. Note that a leaf component is a bridge to its parent. Thus, removing a leaf component decreases the number of nontrivial bridges by one. Suppose the component C is an m-bond with vertices $\{x, y\}$. Using a concurrent write and the fact that we maintain for C a list of all its bridges (and whether or not they are trivial), we are able, in unit time, to determine

that C is a leaf. To remove C from G, simply remove the trivial bridges of C from G, leaving x and y in G and adding to G a new virtual edge from x to y. The data structures are also updated as described above.

Suppose that C is a proper component. It is a leaf when it is associated with at most one 3-connected separating pair. Thus, one can test, in unit time, whether or not C is a leaf. If C is common to no 3-connected separating pairs, then simply ignore C, and do nothing to G or C. However, C is removed from all the other data structures. Suppose that C is common to one pair $\{x, y\}$. To remove C from G: (1) remove all vertices in C except x and y, (2) remove all edges with both end points in C except those between x and y, and (3) add a virtual edge in G from x to y.

To finish our discussion of RAKE, cycles will be considered. Suppose that C is a cycle. Since the vertices on C are unknown, they will be computed "on the fly." Suppose further that $(x_1, \cdots, x_k)$ are the vertices of a cycle component C in the order in which they appear on the cycle. The component C is a leaf if (1) each pair (x_i, x_{i+1}) for $1 \leq i < k$ contains exactly one bridge and that bridge is trivial; and (2) the pair (x_k, x_1) contains at least a trivial bridge. In other words, there exists an adjacent pair of vertices $\{x, y\}$ with a nontrivial bridge that consists of a path. The time required to remove each cycle component that is a leaf seems to require time logarithmic in the length of the its path to detect. We will show how to amortize this cost to achieve an overall time of $O(\log n)$. The edges (x_i, x_{i+1}) for $1 \leq i < k$ form a chain of free edges. Our idea is simply to "compress" these chains of free edges either by the deterministic or by the randomized methods discussed in [29]. In general, any chain can be compressed. Note that a chain of length two may be replaced by a chain of length one, which was formally not free, but it shall be considered free anyway. In this case, the cycle C has been "compressed" to a cycle of size two, a free edge common to a 3-connected separating pair $\{x, y\}$, and a virtual edge from x to y.

Thus, RAKE for cycles consists of compressing chains and removing free edges associated with a 3-connected virtual edge, and then replacing them with a new virtual edge. Other than this timing analysis, we have described RAKE.

The COMPRESS operation is very similar to RAKE. Here, each *proper* and *cycle* component associated with exactly two m-bonds is removed. Suppose that C is a *proper* component associated with 3-connected separating pairs $\{x, y\}$ and $\{z, w\}$. If x, y, z, and w are distinct, then the construction is very similar to the RAKE case. If, on the other hand, $y = w$, the situation is slightly more complicated, since simply removing the edges of C will not separate G. To remove C from G: (1) remove all vertices in C except x, y, z, and w; (2) remove all edges with both end points in C except those between x and y or between z and w; and (3) add a virtual edge in G from x to y and one from z to w.

Suppose C is as described above, except that it is a cycle component. C is removed only when it is a four-cycle component for the case when x, y, z, and w are distinct, or a three-cycle component for the case when $y = w$.

CONTRACT decomposes G into a tree T of 3-connected components after $O(\log n)$ applications. CONTRACT as defined (at least for the sake of analysis) can be viewed as simply CONTRACTION on trees of unbounded degree where RAKE is performed only by combining consecutive children. A case of CONTRACTION very similar to this was analyzed in Theorem 2.10 and shown to require only $O(\log n)$ steps.

Thus, G can be decomposed into a tree of 3-connected graphs, simple cycles, and m-bonds in $O(\log n)$ time using $n^{O(1)}$ processors. This can be stated in the following theorem.

THEOREM 3.2. *The tree of 3-connected components is constructible in $O(\log n)$ time, using $n^{O(1)}$ processors.*

Note that decomposition has been described only where the graph is 2-connected. In general, one must first decompose the graph into a tree of 2-connected components, which will consist of isolated vertices and 2-connected graphs. Second, one must further decompose a 2-connected graph into a tree of 3-connected components.

Ja'Ja' and Simon [18] tested whether or not a 3-connected graph is planar and, if it is, it constructs its planar embedding. However, the construction of a planar embedding for general planar graphs was an open question.

The next section shows how to construct the embedding of a planar graph given the tree of 3-connected components, and how to construct the embedding of each component by viewing it as a tree contraction problem. In this section, we will also define what we mean by "oriented embedding," and will show how to construct planar embeddings that will be used in isomorphism testing.

4. Graph embeddings and some applications.

We will use the following combinatorial definition of an embedding, which is amenable to implementation on a machine.

DEFINITION 4.1. Let $G = (V, E)$ be an undirected graph. Two *darts*, (x, y) and (y, x), are associated with each edge, $e = \{x, y\}$. The vertex x is the *tail* and y is the *head* of the dart (x, y). The graph G is *oriented* by fixing a permutation ϕ of the darts which sends tails to tails and cyclically permutes darts with the same tail. Let $\mathbf{R}$ be the permutation of the darts sending (x, y) to its reflection (y, x). A planar embedding of G can be specified by an orientation of G. See [26], for example. In Fig. 3 we give a small example.

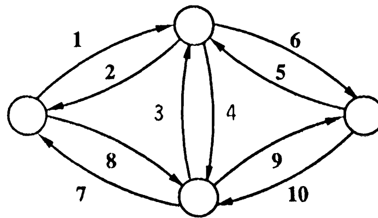


FIG. 3. *A graph with four vertices embedded in the plane. The permutation that determines the orientation at the vertices is* $\phi = (1\,8)(2\,6\,4)(3\,9\,7)(5\,10)$, *written in cycle notation. The reflection of the edges is* $R = (1\,2)(3\,4)(5\,6)(7\,8)(9\,10)$ *and face boundary written as a permutation is* $\phi^* = (1\,6\,10\,7)(2\,8\,3)(4\,9\,5)$.

This definition of a combinatorial embedding is similar to ones described in [10] and is sometimes called an Edmonds embedding; see also [26]. The importance of this definition of embedding is that it is both very simple to understand and easy to represent on a machine. For example, the faces are given by the permutation $\phi^* = \phi \cdot R$. The orbits of ϕ^* are the *faces* of the embedding. Also note that a permutation is stored as an array; thus, most operations on the permutation can be performed in constant time using a linear number of processors.

Ja'Ja' and Simon [18] give a parallel algorithm which constructs a planar embedding of a triconnected planar graph as defined above (note that they call this *planar mesh embedding.* They also construct a straight line embedding, which they call a barycentric embedding, which we do not use).

Using *Parallel Tree Contraction* proves the following theorem.

THEOREM 4.2. *Given the planar embeddings of the* 3*-connected components of a graph* G*, one can compute a planar embedding of* G *in* $O(\log^2 n)$ *time, using* $O(|V|)$ *processors.*

Proof. As described in Part 1 of this paper [29], *Parallel Tree Contraction* can be run "backwards" in an expansion mode which is called *Parallel Tree Expansion.* Here the 3-connectivity algorithm is run in the expansion mode. Thus, one initially starts with a collection of isolated components. The embedding of the isolated graphs is simply the embedding of the individual components. The inverse operation to both RAKE and COMPRESS, in this case, is simply combining two embedded graphs, T and T', by identifying two copies of a virtual edge $\{x, y\}$. The order in which the identification is performed is determined by *Parallel Tree Contraction.* Thus, the only procedure that needs to be shown is how to obtain the embedding for the new graph. Suppose embeddings of T and T' are both common to a virtual edge $e = (x, y)$. Here, the cyclic permutation of T at x is combined with the cyclic permutation of T' at x, which is done by determining a face F of the embedding of T which contains both x and y, and then determining a face F' of the embedding of T' which contains both x and y. The new cyclic order around x will begin by enumerating the darts of x in T as they appear in the embedding of T, starting with the dart in F, and then enumerating the darts of x in C as they appear in the embedding of T', starting with the dart in F'. At the same time, the cyclic permutations of T are combined at y, and the cyclic permutations of T' are combined at y in the same way (see Fig. 4).

To see that this construction can be performed in unit time we write out the permutation explicitly. Since T_1 and T_2 are disjoint graphs, we view them as having a single embedding ϕ. Let $\phi^* = \phi \cdot R$ be its dual. It will suffice to show how to construct the dual embedding $\widehat{\phi}^*$ for the identified graph. For simplicity we leave both copies of the virtual edge in the graph and embed them as parallel edges. The parallel edges can at a later time be removed. Let e_1 be an arc in T_1 from x to y and e_2 be an arc in T_2 from y to x. The dual embedding is as defined below:

$$\widehat{\phi}^* = \begin{cases} e_2 & \text{if } e = e_1, \\ e_1 & \text{if } e = e_2, \\ \phi^*(e_2) & \text{if } \phi^*(e) = e_1, \\ \phi^*(e_1) & \text{if } \phi^*(e) = e_2, \\ \phi^*(e) & \text{otherwise.} \end{cases} \qquad \square$$

This gives the following corollary.

COROLLARY 4.3. *A planar embedding of a planar graph with* n *vertices is constructible in* $O(log^2 n)$ *time, using* $n^{O(1)}$ *processors.*

4.1. Canonical forms of oriented graphs. Whitney [39] has shown that every 3-connected planar graph has exactly two planar embeddings: an embedding ϕ and its reflection ϕ^{-1}. Ja'Ja' and Simon [18] have shown that a planar embedding can be constructed in $O(log^2 n)$ time on a PRAM for a 3-connected planar graph. Any isomorphism of a planar 3-connected graph must preserve its planar orientation up to reflection. More formally, two oriented graphs, (G, ϕ) and (G', ϕ'), are *isomorphic* if there exists a bijective map f from the darts of G to the darts of G' which preserves

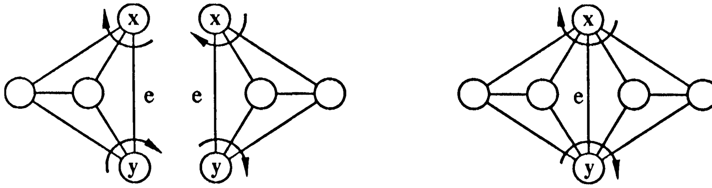


FIG. 4. *Combining the embeddings of two components to get a common embedding.*

both adjacency and orientation: $R'f = fR$ and $\phi' f = f\phi$. Using Whitney's theorem, two 3-connected planar graphs, G' and G, are isomorphic if and only if (G', ϕ') is isomorphic to (G, ϕ) or (G, ϕ^{-1}).

Note that an isomorphism of one embedded graph onto another embedded graph is determined by the image of a single dart. Given a sequence of k numbers, $u = (u_1, \cdots, u_k)$, and a dart e, there exists a unique path of length k, $e = e_0, \cdots, e_k$, where $e_i = \phi^{u_i} R(e_{i-1})$ for $1 \leq i \leq k$. Note that the length is the number of vertices on the path, and *no* is the number of edges. Given a path of darts, a unique sequence of integers can be constructed by choosing the minimum $u_i \geq 0$ such that $e_i = \phi^{u_i} R(e_{i-1})$. Next, we will show how to compute canonical sequences that will be used to compute the canonical forms for embedded graphs.

THEOREM 4.4. *Canonical numbering for oriented graphs is computable in* $O(\log n)$ *time, using* $n^{O(1)}$ *processors.*

A canonical form $M(e)$ for each dart e can be constructed in (G, ϕ). One simply picks the lexicographically least such form. For each dart, $e' \neq e$, the lexicographically least number sequence over the shortest paths from e to e' are found. Suppose that the graph G has d darts. Consider a $d \times d$ matrix where each entry is a number sequence or is blank. Here, the basic scalar operations will be *lexicographically minimum* and *concatenation*, which replace the operations, $+$ and $\times$. Initially, one starts with the matrix containing all paths of length two by storing a sequence of numbers of length one. A matrix product over *minimum* and *concatenation* can be computed in $O(1)$ time using $d^{O(1)} = n^{O(1)}$ processors by Lemma 2.8. Computing $O(\log n)$ iterated powers of this matrix, up to the d power of the original matrix, yields the lexicographically minimum of all shortest paths between all pairs of vertices. Thus, a canonical matrix $M(e)$ is obtained for each dart e in (G, ϕ). The minimum canonical matrix $M(e)$ (under lexicographical order) will be a canonical form for the embedded graph (G, ϕ).

Note that there is an isomorphism if and only if the matrices $M(e)$, as described above, are equal. By also constructing the adjacent matrices for the reflection (G, ϕ^{-1}) and computing the minimum over the larger set of matrices, canonical forms for embedded graphs have been constructed up to reflections. Using the additional fact

that one can compute a planar embedding for a 3-connected graph in $O(log^2 n)$ time on $n^{O(1)}$ PRAM processors, the following theorem is derived from the above.

THEOREM 4.5. *Canonical numbering of* 3*-connected planar graphs can be done in* $O(\log^2 n)$ *time using* $n^{O(1)}$ *PRAM processors.*

5. Reducing the problem of finding canonical forms of planar graphs to the 3-connected case. In this section we give an $O(\log n)$ time reduction from finding canonical forms for general graphs to that of finding canonical forms for 3-connected graphs.

The term "computing canonical forms" means that an oracle accepts as input a 3-connected graph with labels on its darts and vertices and returns an incidence matrix unique up to isomorphism; i.e., it returns canonical linear ordering of the vertices. We also assume that there is a list of new labels that can be added to the darts or vertices.

By using the methods in the last section, one can find up to isomorphism a unique decomposition of a graph into a tree of 3-connected components. In this section, the 3-connected components are simply called "components." Two components are related if one identifies either (1) a virtual edge with orientation (a dart) in one with a virtual edge with orientation in the other, or (2) a vertex in one with a vertex in the other. We will discuss the case where the identifications are edges; i.e., the graph is 2-connected. The general case is a straightforward generalization.

Recall that not all components in a tree of 3-connected components are 3-connected; in particular, they can be either a 3-connected graph, a simple cycle, an m-bond, or an isolated vertex. Canonical forms for these latter graphs can easily be constructed in $O(\log n)$ time.

LEMMA 5.1. *The canonical form for labeled cycles and* m*-bonds can be constructed in* $O(\log n)$ *time using* $n^{O(1)}$ *processors.*

A node of maximum height (at the center of the tree) in a tree of 3-connected components can be found in $O(\log n)$ time by tree contraction, [29]. If the center of the tree is an edge, simply introduce a 2-bond, which will become the center of the tree, as a new component. Thus, without loss of generality, we may assume that the tree is rooted at either a 3-connected component, a virtual edge, or a 2-bond.

Since the rooted tree of 3-connected components is unique up to isomorphism, the vertices shall be ordered into blocks according to the component to which they belong. The separating pair is in the same block with the parent component. The blocks are ordered in postorder (see [36]). However, the children of a component must first be ordered. As in our construction for canonical orderings for regular trees, children will be first ordered at the time when labeled. The characteristic that distinguishes this from a regular tree case is the fact that the children are coupled to their parent by an edge and not a vertex. Thus, more information about the children must be passed to the parent.

Let C be a component and $e = (x, y)$ be the virtual edge of C common to its parent. The edge e is written as two darts d_1 and d_2 (the reverse of d_1). If C is a leaf and a proper component, then, by labeling either d_1 or d_2 with a new label, one gets two labels, L_1 and L_2, respectively, for C. Note that $L_1 = L_2$ if and only if there is an automorphism sending d_1 to d_2. Thus, RAKE is implemented in a straightforward way: (1) compute the labels L_1 and L_2, (2) use the label of each leaf C to label the corresponding darts in the parent of C, and (3) remove C. These labels for C also give us the ordering of the vertices in C, excluding $\{x, y\}$. If $L_1 > L_2$, then use the ordering from L_1; the case is similar if $L_2 > L_1$. On the other hand,

if they are equal, then both orderings are the same, and it does not matter which one is picked. This completes the discussion of RAKE. Note that this computation of RAKE can be executed in unit time, given an oracle for generating the labels L_1 and L_2. COMPRESS will be discussed next.

Let C be a component of degree two where darts e_1 and e_2 are common to the parent and darts d_1 and d_2 are common to the only child. Using two new labels, L and L', assigning L to either e_1 or e_2, and assigning L' to either d_1 or d_2 yields four labelings of C. Use the labeling with maximum value to determine the order of the vertices in C, excluding the end vertices of e. As before, if two labels are equal, then C has a symmetry; either order is the same up to isomorphism. This completes our discussion of COMPRESS.

It is important to point out that we have not determined where, in the final ordering, a given vertex was mapped, since finding this map was not required. This lack of information occurred when one of several orderings for a given component was arbitrarily picked in COMPRESS, and when the children of a component were simply sorted by label. One can determine up to a permutation of order two the ordering of components by using a tree expansion phase [29].

To compute the image of each vertex in the new ordering, it will suffice to determine the orientation induced on the virtual edges by the new ordering; i.e., is a given virtual edge left alone or is it reflected in the new ordering? COMPRESS will be discussed here (the case of RAKE is very similar). Let C be a component with two virtual edges e and d. The possible symmetries consist of reflecting e and independently reflecting d, the Klein 4 group K_4. The actual symmetries will be one of five possible subgroups. Thus, the canonical orderings will be a coset of one of these groups. There are thirteen such cosets, which can be determined by using a parallel call to the oracle for proper components (by applying Theorem 4.4) or which can be determined directly for cycles or m-bonds (by applying Lemma 5.1).

To implement COMPRESS, one need only compute the coset of canonical orderings for a consecutive pair of components from the coset of canonical orderings for each component. Let C and C' be two consecutive components of degree two with virtual edges, d, e, and f, respectively. Further, let A and B be the cosets of canonical orderings of C and C', respectively. Note that A acts on $\{d, e\}$ and B acts on $\{e, f\}$; one wants to return an appropriate coset acting on $\{d, f\}$. If the natural intersection is not empty, it will be returned as the coset of the canonical ordering for C and C'. It will be empty when A and B fix e in opposite orientations. In this case, the coset of the canonical orderings for C and C' consists of a cross-product pair. One acts on d according to A and acts on f according to B. Thus, a method for computing the coset of canonical orderings for the virtual edges of $C \cup C'$ has been presented which uses $O(\log n)$ time and $n^{O(1)}$ processors.

In summary, the CONTRACTION phase consists of the following steps:

1. Compute the canonical labels for all components with degree one or two and determine the coset of canonical orderings on their virtual edges.
2. For leaves, pass the canonical label to the parent.
3. For chains, combine pairs of components as described above, computing both canonical labels and cosets of canonical orderings.

Note that there will be missing cosets when we execute chain contraction. After the tree of components has been reduced to a single component, we perform a tree expansion phase as described in [29] to compute the missing cosets from this further information obtained. Each step can be executed in unit time and thus, by the analysis

in [29], the total time is $O(\log n)$.

We have just given an $O(\log n)$ time reduction from finding canonical forms for general graphs to that of canonical forms for 3-connected components.

$O(\log^2 n)$ time, $n^{O(1)}$ processor algorithms used for finding canonical forms for 3-connected graphs have already been presented. This reduction implies an $O(\log^3 n)$ time, $n^{O(1)}$ processor algorithm that can be used to compute canonical forms for all planar graphs. We summarize our results as a theorem.

THEOREM 5.2. *The problem "Computing canonical forms for a general graph" is $O(\log n)$ time using $n^{O(1)}$ processors reducible to the problem "computing canonical forms for its 3-connected components."*

6. Conclusion. Since the original writing of this paper, many other applications of *Parallel Tree Contraction* have been found. Similarly, many extensions, improvements, and simplifications of the work in this paper have been found. The basic parallel tree contraction can now be done on an EREW PRAM in $O(\log n)$ deterministic time using $n/\log n$ processors, [9], [21], [13], [1]. All of these algorithms use the fact that list-ranking can be performed optimally in deterministic time $O(\log n)$ on an EREW PRAM, [3], [9]. Very simple randomized algorithms for the list-ranking problem are also known, [5]. *Parallel Tree Contraction* can be performed optimally by a randomized algorithm on a parallel model that is more restrictive than an EREW PRAM [4].

In this paper, we restricted our attention to maximal subtree isomorphism. The more general problem for determining if one tree is a subtree of another was first addressed by Matula [22], who gave a polynomial-time algorithm for the problem. A randomized NC algorithm for this problem was given in [14] using the parallel tree contraction technique.

PTC has also been used for efficient parallel evaluation of arithmetic circuits. Prior to the parallel tree contraction technique, the best algorithms for the circuit problem used divide-and-conquer, [37]. Using PTC, one can evaluate circuits on-line in the same time and size as [37] achieved off-line [28], [23].

PTC can be used to design efficient parallel algorithms for problems where the tree is known only implicitly. Examples of such problems occur in the context-free language parsing, constructing Huffman codes, and optimal binary search trees. See [33] for an example of a divide-and-conquer algorithm for such problems and see [6] for a PTC-based approach.

Other applications include: testing triconnectivity of a graph [27], [11]; testing graph planarity [19]; finding separator for planar graphs [25], [12]; and finding algorithms for reducible flow graphs [30].

This is not an exhaustive list, and we apologize for the works which we have neglected to reference.

REFERENCES

[1] K. ABAHAMSON, N. DADOUN, D. K. KIRKPATRICK, AND T. PRZYTYCKA, *A simple parallel tree contraction algorithm (preliminary version)*, in Proc. 25th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, September/October 1987, pp. 624–633.

[2] M. AJTAI, J. KOMLOS, AND E. SZEMEREDI, *An $O(n \log\ n)$ sorting network*, in Proc. 15th Annual Symposium on the Theory of Computing, 1983, pp. 1–9.

[3] R. J. ANDERSON AND G. L. MILLER, *Deterministic parallel list ranking*, in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988,

Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, pp. 81–90.

[4] R. J. ANDERSON AND G. L. MILLER, *Optical communication for pointer based algorithms*, Tech. Report CRI 88-14, Department of Computer Science, University of Southern California, Los Angeles, CA, 1988.

[5] ———, *A simple randomized parallel algorithm for list-ranking*, Inform. Process. Lett., 33 (1990), pp. 269–273.

[6] M. ATALLAH, R. KOSARAJU, L. LARMORE, G. L. MILLER, AND S.-H. TENG, *Constructing trees in parallel*, in Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, June 1989, pp. 421–431.

[7] I. BAR-ON AND U. VISHKIN, *Optimal parallel generation of a computation tree form*, ACM Trans. Programming Languages and Systems, 7 (1985), pp. 348–357.

[8] R. COLE, *Parallel merge sort*, in Proc. 27th IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1987, pp. 511–516.

[9] R. COLE AND U. VISHKIN, *Optimal parallel algorithms for expression tree evaluation and list ranking*, in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 91–100.

[10] J. EDMONDS, *A combinatorial representation for polyhedral surfaces*, Amer. Math. Soc., 7 (1960), p. 646.

[11] D. FUSSELL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacement*, in Proc. Internat. Conference on Automata, Languages and Programming, 1989, Springer-Verlag, pp. 379–393.

[12] H. GAZIT AND G. L. MILLER, *A parallel algorithm for finding a separator in planar graphs*, in 28th IEEE Annual Symposium on Foundations of Computer Science, Los Angeles, CA, October 1987, pp. 238–248.

[13] H. GAZIT, G. L. MILLER, AND S.-H. TENG, *Optimal tree contraction in an* EREW *model*, in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., Plenum Press, New York, 1988, pp. 139–156.

[14] P. B. GIBBONS, R. M. KARP, G. L. MILLER, AND D. SOROKER, *Subtree isomorphism is in random* NC, in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing, (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, 1988, pp. 43–52.

[15] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fourth Edition, Oxford University Press, Oxford, U.K., 1959.

[16] J. E. HOPCROFT AND R. E. TARJAN, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.

[17] O. H. IBARRA AND S. MORAN, *Probabilistic algorithms for deciding equivalence of straight-line programs*, J. Assoc. Comput. Mach., 30 (1983), pp. 217–228.

[18] J. JA'JA' AND J. SIMON, *Parallel algorithms in graph theory: Planarity testing*, SIAM J. Comput., 11 (1982), pp. 314–328.

[19] P. KLEIN AND J. REIF, *An efficient parallel algorithm for planarity*, J. Comput. System Sci., (1988), pp. 190–246.

[20] D. E. KNUTH AND L. TRABB PARDO, *Analysis of a simple factorization algorithm*, Theoret. Comput. Sci., 3 (1976), pp. 321–348.

[21] S. R. KOSARAJU AND A. L. DELCHER, *Optimal parallel evaluation of tree-structured computation by ranking* (*extended abstract*), in VLSI Algorithms and Architectures: Third Aegean Workshop on Computing (AWOC 88), June/July 1988, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, New York, pp. 101-110.

[22] D. W. MATULA, *Subtree isomorphism in* $O(n^{5/2})$, Ann. Discrete Math., 2 (1978), pp. 91–106.

[23] E. W. MAYR, *The dynamic tree expression problem*, in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., 1988, Plenum Press, New York, pp. 157–180.

[24] G. L. MILLER, *Riemann's hypothesis and tests for primality*, J. Comput. System Sci., 13 (1976), pp. 300–317.

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

[26] ———, *An additivity theorem for the genus of a graph*, J. Combin. Theory, Ser. B, 43 (1987), pp. 25–47.

[27] G. L. MILLER AND V. RAMACHANDRAN, *A new graph triconnectivity algorithm and its parallelization* (*extended abstract*), in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, May 1987.

[28] G. L. MILLER, V. RAMACHANDRAN, AND E. KALTOFEN, *Efficient parallel evaluation of straight-line code and arithmetic circuits*, SIAM J. Comput., 17 (1988), pp. 687–695.

[29] G. L. MILLER AND J. H. REIF, *Parallel tree contraction Part* 1 : *Fundamentals*, in Randomness and Computation, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 47–72.

[30] V. RAMACHANDRAN, *Fast parallel algorithms for reducible flow graphs*, in Concurrent Computations: Algorithms, Architecture and Technology, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds., Plenum Press, New York, 1988, pp. 117–138.

[31] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, in Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Boston, 1983, pp. 10–16.

[32] J. B. ROSSER AND L. SCHOENFIELD, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 64–94.

[33] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[34] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[35] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.

[36] R. E. TARJAN AND U. VISHKIN, *Finding biconnected components and computing tree functions in logarithmic parallel time*, in Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science, FL, 1984, pp. 12–22.

[37] L. G. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641–644.

[38] U. VISHKIN, *Randomized speed-ups in parallel computation*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Washington D.C., April 1984, Association for Computing Machinery, pp. 230–239.

[39] H. WHITNEY, *A set of topological invariants for graphs*, American J. Math., 55 (1937), pp. 321–335.

[40] J. C. WYLLIE, *The complexity of parallel computations*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1981.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1148–1156, December 1991
© 1991 Society for Industrial and Applied Mathematics
010

ON SETS WITH EFFICIENT IMPLICIT MEMBERSHIP TESTS*

LANE A. HEMACHANDRA† AND ALBRECHT HOENE‡

Abstract. This paper completely characterizes the complexity of implicit membership testing in terms of the well-known complexity class OptP, optimization polynomial time, and concludes that many complex sets have polynomial-time implicit membership tests.

Key words. computational complexity, implicit membership testing, optimization polynomial time

AMS(MOS) subject classifications. 68Q15, 03D15, 68Q10

1. Introduction. Deterministic polynomial time, P, is one notion that has been proposed as loosely defining the sets that are efficiently computable [11], [12]. By definition, for sets in P one can quickly check whether a given element is a member. Unfortunately, not all sets of interest are in P.

For sets outside of P, it seems natural to ask which properties can be quickly computed, and, conversely, for properties slightly less revealing than membership, it seems natural to characterize the sets for which that property can be quickly computed. Indeed, the study of exactly which sets are simple under various operations other than membership testing is an emergent theme in theoretical computer science. Though it has long been known that many complex problems can be efficiently approximately solved, current research efforts show that fundamental algorithmic operations—data compression, perfect hashing, and enumeration—can be efficiently performed on many complex sets [20], [16], [21]. The present paper shows that even sets of extremely high complexity may have polynomial-time algorithms for implicitly testing membership.

The recent work of Goldsmith, Hemachandra, Joseph, and Young [15] can be viewed from the perspective of this paradigm of finding efficient operations for complex sets. Goldsmith, Joseph, and Young [17] defined and extensively studied the behavior of near-testable (NT) sets—those sets L for which on input x one can quickly compute which of (a) $(x \in L) \oplus (x_- \in L)$ or (b) $\mathrm{NOT}[(x \in L) \oplus (x_- \in L)]$ holds.[1] That is, one can quickly test whether exactly one of an element and its predecessor are in the set. [15] showed that NT is essentially the same as the class $\oplus$P, "parity polynomial time" [28], [14].

In this paper, we consider a property that in some sense combines aspects of P and NT. Fix a set L. It is clear that for each $x \neq \epsilon$, exactly two of the following four statements hold:

(1) $x \in L$,
(2) $x \notin L$,
(3) $(x \in L) \oplus (x_- \in L)$,
(4) $\mathrm{NOT}[(x \in L) \oplus (x_- \in L)]$.

* Received by the editors Novemeber 23, 1989; accepted for publication (in revised form) January 23, 1991.

† Department of Computer Science, University of Rochester, Rochester, New York 14627. The research of this author was supported in part by National Science Foundation grants CCR-8996198 and CCR-8957604.

‡ Fachbereich Informatik, Technische Universität Berlin, D-1000 Berlin 10, Germany. The research of this author was supported in part by a Deutsche Forschungsgemeinschaft Postdoktorandenstipendium.

[1] $\oplus$ represents "exclusive or," and x_- represents the predecessor of x in standard lexicographical order.

If we could determine in polynomial time which of the third or fourth of the above statements (henceforth 3/4) holds, L would be in NT. If we could determine in polynomial time which of the first or second of the above statements (henceforth 1/2) holds, L would be in P. Clearly, of 1/2, exactly one holds, and of 3/4, exactly one holds. Thus, even though we know that for each x exactly two of 1/2/3/4 hold, if we could compute which two hold in polynomial time, then L would be in P.

Nearly near-testable sets (NNT) allow us to capture a more modest amount of information about a given element x. We define a set to be nearly near-testable if for each x, we can find, in polynomial time, *one* of 1/2/3/4 that holds. More formally, we have Definition 1.1.

DEFINITION 1.1. A language L is in NNT if there is a polynomial-time computable function f such that for each x, either:

- $(f(x) = \text{“}x \in L\text{”})$ and $(x \in L)$, OR
- $(f(x) = \text{“}x \notin L\text{”})$ and $(x \notin L)$, OR
- $(f(x) = \text{“}(x \in L) \oplus (x_- \in L)\text{”})$ and $((x \in L) \oplus (x_- \in L))$, OR
- $(f(x) = \text{“NOT}[(x \in L) \oplus (x_- \in L)]\text{”})$ and $(\text{NOT}[(x \in L) \oplus (x_- \in L)])$.

When one nearly near-tests an element of a nearly near-testable set, one obtains partial information that does not, in general, suffice to immediately compute membership; nonetheless, with an exponential number of tests, one can (trivially) recover membership information. Thus, NNT sets have *implicit* membership tests. The information is there, but the cost of extraction is high. It follows immediately from the brute force testing just referred to that NNT is contained in PSPACE.

Clearly, NT $\subseteq$ NNT. This paper shows that, just as NT has been shown to be related to $\oplus$P [15], so also is NNT related to the optimization complexity class OptP [23], [9]. In particular, Theorem 1.5 says that NNT is the same (within the flexibility of $\leq^p_{1-1}$ reductions) as $\oplus$P altered by allowing an OptP function as a second argument to the underlying nondeterministic TM.[2]

DEFINITION 1.2 ([23]).

(1) An NP *metric Turing Machine*, $\widehat{N}$, is a nondeterministic polynomial-time Turing machine such that every branch writes a binary number and accepts; for $x \in \Sigma^*$ we write $\text{opt}^{\widehat{N}}(x)$ for the largest value on any branch of $\widehat{N}$ on input x.

(2) A function f is in OptP (optimization polynomial time) if there is an NP metric Turing machine $\widehat{N}$ such that:

$$f(x) = \text{opt}^{\widehat{N}}(x) \quad \text{for all } x \in \Sigma^*.$$

DEFINITION 1.3 ([26]). $count_N(w_1)$ ($count_N(w_1, w_2)$) represents the number of accepting paths of machine N running on input w_1 (on input w_1, w_2).

DEFINITION 1.4. L is in $\oplus$OptP if and only if there is a nondeterministic polynomial-time Turing machine N and a function $f \in$ OptP such that:

$$x \in L \iff count_N(f(x)) \text{ is odd.}^3$$

THEOREM 1.5. (1) NNT $\subseteq$ $\oplus$OptP.

[2] Throughout this paper, the maximum function is always applied to sets of strings (which are in fact integers encoded in binary), and returns the integer value that the lexicographically largest string encodes; we adopt the convention that $\max(\emptyset) = 0$.

[3] Lemma 2.1 of the next section proves that $\oplus$OptP remains unchanged when this condition is replaced with the perhaps more natural condition: $x \in L \iff count_N(x, f(x))$ is odd.

(2) $\oplus$OptP $\leq_m^p$ NNT; *indeed,* $\oplus$OptP $\leq_{1-1}^p$ NNT. *That is, for each language L in* $\oplus$OptP, *there is a language L' in* NNT *such that L reduces to L' via a one-to-one reduction computable in deterministic polynomial time.*

This paper's proof of Theorem 1.5 extends the techniques of [17] and [15] by showing that one can construct a way for the maximization performed by the OptP functions within $\oplus$OptP sets to be encoded in the instant jackpot options (options 1/2) of some nearly near-testable set.

Finally, by proving that with probability one relative to a random oracle A, $\mathrm{NT}^A \neq \mathrm{NNT}^A$, we suggest that NT may differ from NNT. Probability-one separations of pairs of classes in this range of complexity are usually attempted via circuit techniques, or via the techniques, now believed to be invalid [6], used in the Bennett–Gill probability-one separation of $\oplus$P and PP [7]. In contrast, we prove our new result by a novel and simple approach: we "reduce" the probability-one separation of these two complex classes to a well-understood task—that of separating a parity-like language from P with probability one.

2. Results. This section shows that implicit membership testing is closely related to the optimization class OptP.

We start with a preliminary lemma that both shows the robustness of $\oplus$OptP—$\oplus$OptP can be defined in terms of OptP, Δ_2^p, or maximization—and also translates $\oplus$OptP into a form that will be used in the proof of Theorem 1.5. In the following lemma, parts 2 and 3 can be shown to be equivalent using the notions developed in [23] (see also [27]). The equivalence of parts 1 and 4 establishes the version of $\oplus$OptP to be used in the proof of Theorem 1.5. Just as parts 1 and 2 are related by showing that the "x" argument is superfluous, so also could one add to the lemma below new parts 3′ and 4′, by replacing "$count_N(x, \cdots)$" with "$count_N(\cdots)$."

LEMMA 2.1. *The following are equivalent:*

1. *L is in* $\oplus$OptP.
2. *There is a nondeterministic polynomial-time Turing machine N and a function $f \in$* OptP *such that:*

$$x \in L \iff count_N(x,\, f(x)) \text{ is odd.}$$

3. *There is a nondeterministic polynomial-time Turing machine N and a function f computable by a* P^{NP} *machine (i.e., in* $\mathrm{FP}^{\mathrm{NP}}$, *in the notation of [23]) such that:*

$$x \in L \iff count_N(x,\, f(x)) \text{ is odd.}$$

4. *There is a nondeterministic polynomial-time Turing machine N, a polynomial $r(\cdot)$, and a polynomial-time computable predicate $R(\cdot,\, \cdot)$, such that:*

$$x \in L \iff count_N(x,\, \max_{|z|=r(|x|)} \{z \mid R(x,\, z)\}) \text{ is odd.}$$

We defer the proof of Lemma 2.1 until after that of Theorem 1.5.

THEOREM 1.5 (1) NNT $\subseteq \oplus$OptP.

(2) $\oplus$OptP $\leq_m^p$ NNT; *indeed,* $\oplus$OptP $\leq_{1-1}^p$ NNT. *That is, for each language L in* $\oplus$OptP, *there is a language L' in* NNT *such that L reduces to L' via a one-to-one reduction computable in deterministic polynomial time.*

Due to the fact that $\oplus$OptP (but not necessarily NNT) is closed downwards under $\leq_{1-1}^p$, we immediately obtain the following corollary.

COROLLARY 2.2. (1) *The downward closures of* NNT *and* $\oplus$OptP *under* $\leq_m^p$ *are identical. That is,* $\{L \mid (\exists L' \in \mathrm{NNT})[L \leq_m^p L']\} = \{L \mid (\exists L' \in \oplus\mathrm{OptP})[L \leq_m^p L']\}$.

(2) *The downward closures of* NNT *and* $\oplus$OptP *under* $\leq_{1-1}^p$ *are identical.*

It follows immediately from Theorem 1.5, the obvious inclusions of Proposition 2.3, and Toda's [30] recent results,[4] that it is extremely unlikely that nearly near-testable sets have efficient membership tests.

PROPOSITION 2.3. $\oplus\mathrm{P} \subseteq \oplus\mathrm{OptP}$ *and* $\mathrm{NP} \subseteq \oplus\mathrm{OptP}$.

COROLLARY 2.4. (1) *If* $\mathrm{NNT} = \mathrm{P}$, *then* $\mathrm{P} = \mathrm{NP} = \mathrm{PH} = \oplus\mathrm{P}$.

(2) *If* $\mathrm{NNT} \subseteq \Sigma_k^p$, *then* $\Sigma_{k+1}^p = \mathrm{PH}$.

(3) $\mathrm{PH} \subseteq \mathrm{BP}{\cdot}\mathrm{NNT}$.[5]

Proof of Theorem 1.5. Part (1) is straightforward, via using the maximum to find the largest element less than the current element that yields absolute membership information, and then counting the parity of the number of changes in membership status between it and the input value.

Now, let us show that for each $L \in \oplus\mathrm{OptP}$, there is an $L' \in \mathrm{NNT}$ such that $L \leq_{1-1}^p L'$. Consider an arbitrary set L in $\oplus$OptP. By Lemma 2.1, let N, $r(\cdot)$, and $R(\cdot, \cdot)$ be a machine, polynomial, and predicate for L, in the sense of Lemma 2.1, part 4. Without loss of generality, for all x and m let $N(x, m)$ not have any member of 0^* as an accepting computation path. Without loss of generality, let $r(\cdot)$ be monotonically increasing. Without loss of generality, there is an integer $k \geq 2$ such that for all x and m, all computation paths of $N(x, m)$ are of length exactly $|x|^k + k$,[6] and all paths of this length are present. Without loss of generality, $R(x, 0^{r(|x|)})$ holds for all x.

We define a (nonstandard but) very simple non-onto pairing function. Let x, m, $p \in \{0, 1\}^*$, and let l be an integer greater than zero; define $tweak_l(x, m, p)$ to be xmp (the concatenation, without any separation characters, of x, m, and p) if $|m| = r(|x|)$ and $|p| = |x|^l + l$, and let $tweak_l(x, m, p)$ be undefined otherwise. $tweak_l(x, m, p)$ is one-to-one everywhere it is defined, and given a string z, we can determine in polynomial time whether z is in the range of $tweak_l$, and, if so, what its inverse is.

We define the set L' as follows, using *lex* as a subscript to denote operations performed with respect to the standard lexicographical order:

> If there are no x, m, p such that $z = tweak_k(x, m, p)$, then $z \notin L'$. Otherwise, let x, m, p be the unique strings such that $tweak_k(x, m, p) = z$. Let z be in L' if:

(1) $p = 0^{|x|^k+k}$ and $R(x, m)$, or

(2) there are an odd number of strings p' such that:

   (a) $p' \leq_{lex} p''$, where $p'' = p$ if $R(x, m)$ holds and $p'' = 1^{|x|^k+k}$ if $R(x, m)$ does not hold, and

   (b) $p' >_{lex} 0^{|x|^k+k}$, and

[4] Namely, that (a) $\mathrm{PH} \subseteq \mathrm{BP}{\cdot}\oplus\mathrm{P}$, and (b) if $\oplus\mathrm{P} \subseteq \Sigma_k^p$, then the polynomial hierarchy collapses to Σ_{k+1}^p.

[5] In fact, for the same reason, the stronger result holds that $\mathrm{PH} \subseteq \mathrm{BP}{\cdot}\mathrm{NT}$, where NT is defined as in [17]. The BP operator is defined in [29].

[6] We assume that Turing machines of two inputs are organized so that the machine may read its first argument quickly even when the second argument is long; for example, the inputs might be placed on different tracks of the input tape, or the input tape might contain *input*1#*input*2. Looking at Lemma 2.1, part 4, it is clear that in such a model the fact that $|m|$ does not appear in $|x|^k + k$ is not problematic.

(c) $N(x, \widehat{m})$ has p' as an accepting path, where

$$\widehat{m} = \max_{|j|=|m| \text{ and } j \leq_{lex} m} \{j \mid R(x, j)\}.^7$$

All strings not granted membership in L' by the above rules are not members of L'.

We claim that $L' \in$ NNT, and $L \leq_{1-1}^{p} L'$. Let us show that L' is nearly near-testable by explicitly presenting the polynomial-time procedure for nearly near-testing L' (see Definition 1.1).

Given an input z, check if there are x, m, and p, such that $tweak_k(x, m, p) = z$. If not, print "$z \notin L'$." If such x, m, and p exist (and thus are, perforce, unique), then:

If $R(x,m)$ and $p = 0^{|x|^k+k}$ then print "$z \in L'$";

else if $R(x, m)$ and $p = 0^{|x|^k+k-1}1$ then print "$z \in L'$" if p is an accepting path of $N(x, m)$ and print "$z \notin L'$" if p is a rejecting path of $N(x, m)$;

else if p is an accepting path of $N(x, m)$ and $R(x, m)$ then print "$(z \in L') \oplus (z_- \in L')$";

else print "NOT$[(z \in L') \oplus (z_- \in L')]$."

It is easily seen (by looking at the definition of L') that this polynomial-time procedure nearly near-tests L'. Finally, $x \in L$ if and only if $tweak_k(x, 1^{r(|x|)}, 1^{|x|^k+k})$ is in L'; thus, $L \leq_{1-1}^{p} L'$; this is because $tweak_k(x, 1^{r(|x|)}, 1^{|x|^k+k})$ counts the parity of the number of paths of N when its first input is x and its second input is the true maximum. □

It is clear from the proof that L reduces to L' by a $\leq_{1-1}^{p}$ reduction that is trivially invertible, and whose range is in P. Via straightforward alteration of the given construction, one can show that every $\oplus$OptP set L reduces to some NNT set $\hat{L}$ by a $\leq_{1-1}^{p}$ reduction that is trivially invertible and whose range is Σ^*.

Proof of Lemma 2.1. The equivalence of parts 2 and 3 follows immediately from Theorem 3.1 of [23] ($3 \Rightarrow 2$ is instant; $2 \Rightarrow 3$ by having N assume the role of computing the h function of Krentel's theorem 4 [23]). It is also clear that $4 \Rightarrow 2$, as OptP functions can easily find the maximum value on which a polynomial predicate is true, by having each path guess a value and if the predicate is true, print the value. And it is immediate that $1 \Rightarrow 2$. It is also clear that $2 \Rightarrow 1$. To see this, let L_2 be a language satisfying part 2, via OptP function f_2 and nondeterministic polynomial-time Turing machine N_2. Then $L \in \oplus$OptP via OptP function f_1 and nondeterministic polynomial-time Turing machine N_1, where $f_1(x) = \langle x, f_2(x)\rangle$ and $N_1(y)$ starts by decoding its input into $y = \langle x, z\rangle$ and simulates $N_2(x, z)$. The pairing function must be chosen with care, so as not to interfere with the optimization; the venerable function $\langle x, z\rangle = x_1 0 x_2 0 \cdots x_r 1 z_1 \cdots z_{r'}$ is fine, where x_i (z_i) is the ith bit of x (z).

We now turn to showing that $2 \Rightarrow 4$. Let L be an arbitrary set satisfying part 2 of Lemma 2.1.

Choose machines that certify this; in particular, we will use N_2 to denote the machine N of part 2 of Lemma 2.1, and we will use $\widehat{N}$ to denote the (metric) Turing machine (see Definition 1.2) for the OptP function f of part 2 of Lemma 2.1. Without loss of generality, for some integer k, $\widehat{N}$ has, on inputs of length n, exactly 2^{n^k+k} paths,

[7] Recall footnote 2.

each of length exactly $n^k + k$. We will now define N_4, r_4, and R_4, as in part 4 of Lemma 2.1, in order to show that L satisfies part 4.

Set $r_4(n) = 2(n^k + k)$. Let the predicate $R_4(x, z)$ accept if and only if:

(1) $|z| = 2(|x|^k + k)$ and,

(2) if we view z as $z = concatenation(z_{start}, z_{end})$, $|z_{start}| = |z_{end}|$, then path z_{end} of the computation tree of $\widehat{N}(x)$ prints the integer value that z_{start}, viewed as a binary string, encodes.

Let $N_4(x, y)$ reject if $|y| \neq 2(|x|^k + k)$. Otherwise, $N_4(x, y)$ computes $\widehat{p}$, the value of the first $|x|^k + k$ bits of y viewed as an integer, and then simulates $N_2(x, \widehat{p})$.

Why does this work? The first and second halves of z hold, respectively, the values and paths of the OptP function. The maximization finds the largest value and the path on which it is obtained. N_4 starts by throwing away the superfluous path information (which, in allowing the max to find the value of the OptP function, has already served its purpose), and proceeds to exactly simulate the machine N_2 of part 2 of our lemma. □

The above results give evidence that NNT and ⊕OptP are the same class, given the flexibility provided by many-one (or even one-to-one) polynomial-time reductions. One might naturally ask whether NNT and ⊕OptP are identical. As an anonymous referee has pointed out: if NT = ⊕P, then the polynomial hierarchy (and indeed $\mathrm{PP}^{\mathrm{PH}}$) equals P (combining an observation of [15] with Toda's recent result [30]),[8] and, by the same argument, the following holds.

PROPOSITION 2.5. *If* NNT = ⊕OptP, *then the polynomial hierarchy* (*and indeed* $\mathrm{PP}^{\mathrm{PH}}$) *equals* P.

This paper has studied the class NNT. However, the class NT has been extensively investigated in earlier papers [17], [15]. If NT = NNT, there would be no need for a separate study of NNT; and indeed, at first NNT seems very closely related to NT. In fact, if one looks not at the classes of *polynomial-time* near-testable and nearly near-testable sets, but rather at the classes of sets with *exponential-time*[9] near-testing and nearly near-testing functions, it is immediate that they *are* the same, and both are equal to exponential time. Similar results hold for many other classes, such as for recursive near-testers and recursive nearly near-testers.

Nonetheless, we give evidence that NT and NNT are not the same.

One traditional way of opening the possibility that classes differ is to present a relativized world in which they do differ [4].[10] Somewhat stronger evidence can be presented by showing that classes differ relative to almost every oracle ([7], see also [10], [3], [25]), though even this level of evidence does *not* ensure that the classes are different in the unrelativized world [24]. Our proof proceeds by "reducing" to a simpler task the problem of separating NT from NNT.

THEOREM 2.6. $\mathrm{NT}^A \subsetneq \mathrm{NNT}^A$ *with probability one relative to a random oracle* A.

Proof of Theorem 2.6. By NT^A (NNT^A), we mean the sets that have a near-testing (nearly near-testing) function computable in P^A. Consider the following nearly near-testing function, which will implicitly define the language, L_A, that it nearly

[8] Further discussion of the relationship between NT and ⊕P can be found in [5].

[9] The claim holds for either of the standard definitions of exponential time, $\mathrm{E} = \cup_{c>0} \mathrm{DTIME}[2^{cn}]$ and $\mathrm{EXP} = \cup_{c>0} \mathrm{DTIME}[2^{n^c}]$.

[10] To be conservative about such claims, one should make only the claim that conflicting relativizations show that a problem will not be resolved by relativizable techniques; then one can discuss in detail the extent to which present and possible future techniques are or are not relativizable, keeping in mind Shamir's recent nonrelativizing IP = PSPACE result [18], [1], [19].

near-tests.

$$f_A(x) = \begin{cases} \text{“}x \in L_A\text{”} & \text{if } x \in 0^* \text{ and } x \in A, \\ \text{“}x \notin L_A\text{”} & \text{if } x \in 0^* \text{ and } x \notin A, \\ \text{“}(x \in L_A) \oplus (x_- \in L_A)\text{”} & \text{if } x \notin 0^* \text{ and } x \in A, \\ \text{“NOT}[(x \in L_A) \oplus (x_- \in L_A)]\text{”} & \text{if } x \notin 0^* \text{ and } x \notin A. \end{cases}$$

It is important to note that the definitions of f_A and L_A are not circular, as outputs of f_A—such as "$(x \in L_A) \oplus (x_- \in L_A)$"—are purely syntactic statements, which are made to be true by defining L_A appropriately. Note that (trivially) for all A, $L_A \in \mathrm{NNT}^A$, since there is a nearly near-testing function (namely, f_A).

Now let us ask how likely it is that, relative to random oracle A, the set L_A is in NT^A. Consider an oracle A' such that $L_{A'} \in \mathrm{NT}^{A'}$, and let the near-testing function be called f' ($\in P^{A'}$). Clearly, A' will have an odd number of strings of length n if and only if $1^n \in L_{A'}$. (This is because, in the case where $0^n \notin L_{A'}$, we will have $1^n \in L_{A'}$ only if the second line of the definition of f occurs an odd number of times; in the case where $0^n \in L_{A'}$, we will have $1^n \in L_{A'}$ only if the second line of the definition of f occurs an even number of times, but 0^n itself will contribute to the parity to make the parity odd.) However, 1^n is in $L_{A'}$ if and only if

$$(0^{n+1} \in A') \oplus (f'(0^{n+1}) \text{ prints “}(0^{n+1} \in L_{A'}) \oplus (1^n \in L_{A'})\text{”}),$$

since f' is a near-tester. The test just given is a $\mathrm{P}^{A'}$ test that computes the parity of the number of strings of a given length in A'. That is, we have shown that if $L_{A'} \in \mathrm{NT}^{A'}$, then there is a polynomial-time deterministic Turing machine that on input 1^n computes the parity of the set $\{w \mid n = |w| \text{ and } w \in A\}$. By standard techniques [7],[11] the class of such oracles has measure zero. Thus with probability one relative to a random oracle A, $\mathrm{NNT}^A \neq \mathrm{NT}^A$ (indeed, $\mathrm{NNT}^A \supsetneq \mathrm{NT}^A$, since the fact that $\mathrm{NT} \subseteq \mathrm{NNT}$ relativizes). □

3. Conclusion. This paper shows that NNT is essentially the same as $\oplus$OptP. Thus, the complexity of implicit membership testing is closely related to the complexity of optimization.

We note that it has recently been shown that NT and NNT are related to notions of polynomial enumerability [21], [22], and that probabilistic reductions—whose surprising power has been demonstrated by Toda [30]—relate NT and NNT [22].

One interesting open question concerns the downward closure properties of $\oplus$OptP. Though it is well known that $\oplus\mathrm{P}^{\oplus\mathrm{P}} = \oplus\mathrm{P}$ [28], it is not clear whether $\oplus\mathrm{OptP}^{\oplus\mathrm{OptP}} = \oplus\mathrm{OptP}$, though it is not hard to see that[12] $\mathrm{R}^p_{\textit{truth-table}}(\oplus\mathrm{OptP}) = \oplus\mathrm{OptP}$. Can one show that $\oplus\mathrm{OptP}^{\oplus\mathrm{OptP}} = \oplus\mathrm{OptP}$, or at least that $\mathrm{P}^{\oplus\mathrm{OptP}} = \oplus\mathrm{OptP}$? We suspect that these equalities do not hold.

Acknowledgments. We are grateful to Jin-yi Cai, Judith Goldsmith, Johannes Köbler, Steven Rudich, and Paul Young for helpful conversations, to Richard Beigel for providing an advance copy of [6] and for helpful discussions regarding Proposition 2.5

[11] There has been some confusion in this area recently. Beigel [6] has noted that Bennett and Gill's proof of [7, Thm. 3] is incorrect, and that the result itself may not be true. However, the still-correct techniques of [7], in particular their Lemma 1, easily suffice to show that for random oracle A, one cannot compute the parity function of A in polynomial time relative to A.

[12] $\mathrm{R}^p_{\textit{truth-table}}(\mathcal{C})$ denotes the class of all sets L such that there exists a set $L' \in \mathcal{C}$ that L polynomial-time truth-table reduces to; such classes have attracted some interest recently [8], [2], [13].

and [5], [6]. We thank Andrew Yao for guidance during the refereeing process, and two anonymous referees for thoughtful and helpful comments.

REFERENCES

[1] E. ALLENDER, *Oracles versus proof techniques that do not relativize*, in Proc. 1990 SIGAL International Symposium on Algorithms, Lecture Notes in Computer Science 450, Springer-Verlag, Berlin, New York, August 1990, pp. 39–52.

[2] E. ALLENDER, L. HEMACHANDRA, M. OGIWARA, AND O. WATANABE, *Relating equivalence and reducibility to sparse sets*, SIAM J. Comput., to appear.

[3] L. BABAI, *A random oracle separates* PSPACE *from the polynomial hierarchy*, Inform. Process. Lett., 26 (1987), pp. 51–53.

[4] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the P=?NP question*, SIAM J. Comput., 4 (1975), pp. 431–442.

[5] R. BEIGEL, *Bi-immunity results for cheatable sets*, Theoret. Comput. Sci., 73 (1990), pp. 249–263.

[6] ———, *Relativized counting classes: Relations among thresholds, parity, and mods*, J. Comput. System Sci., 42 (1991), pp. 76–96.

[7] C. BENNETT AND J. GILL, *Relative to a random oracle A*, $P^A \neq NP^A$ *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[8] R. BOOK AND K. KO, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[9] D. BRUSCHI, D. JOSEPH, AND P. YOUNG, *A structural overview of* NP *optimization problems*, Algorithms Review, 2 (1991), pp. 1–26.

[10] J. CAI, *With probability one, a random oracle separates* PSPACE *from the polynomial-time hierarchy*, J. Comput. System Sci., 38 (1989), pp. 68–85.

[11] A. COBHAM, *The intrinsic computational difficulty of functions*, in Proc. 1964 International Congress for Logic, Methodology and Philosophy of Science, North–Holland, Amsterdam, 1964, pp. 24–30.

[12] J. EDMONDS, *Paths, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449–467.

[13] R. GAVALDÀ AND O. WATANABE, *On the computational complexity of small descriptions*, in Proc. 6th Structure in Complexity Theory Conference, IEEE Computer Society Press, June/July 1991, pp. 89–101.

[14] L. GOLDSCHLAGER AND I. PARBERRY, *On the construction of parallel computers from various bases of boolean functions*, Theoret. Comput. Sci., 43 (1986), pp. 43–58.

[15] J. GOLDSMITH, L. HEMACHANDRA, D. JOSEPH, AND P. YOUNG, *Near-testable sets*, SIAM J. Comput., 20 (1991), pp. 506–523.

[16] J. GOLDSMITH, L. HEMACHANDRA, AND K. KUNEN, *On the structure and complexity of infinite sets with minimal perfect hash functions*, Proceedings of the 11th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, to appear.

[17] J. GOLDSMITH, D. JOSEPH, AND P. YOUNG, *Self-reducible,* P*-selective, near-testable, and* P*-cheatable sets: The effect of internal structure on the complexity of a set*, in Proc. 2nd Structure in Complexity Theory Conference, IEEE Computer Science Press, June 1987, pp. 50–59.

[18] J. HARTMANIS, *Solvable problems with conflicting relativizations*, Bull. European Assoc. Theoret. Comput. Sci., 27 (1985), pp. 40–49.

[19] J. HARTMANIS, R. CHANG, D. RANJAN, AND P. ROHATGI, *Structural complexity theory: Recent surprises*, in Proc. 2nd Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 447, Springer-Verlag, Berlin, New York, July 1990, pp. 1–12.

[20] L. HEMACHANDRA, *Algorithms from complexity theory: Polynomial-time operations for complex sets*, in Proc. 1990 SIGAL International Symposium on Algorithms, Lecture Notes in Computer Science 450, Springer-Verlag, Berlin, New York, Aug. 1990, pp. 221–231.

[21] L. HEMACHANDRA, A. HOENE, D. SIEFKES, AND P. YOUNG, *On sets polynomially enumerable by iteration*, Theoret. Comput. Sci., 80 (1991), pp. 203–226.

[22] A. HOENE, *Polynomielle Splinter*, Ph.D. thesis, Fachbereich Informatik, Technische Universität Berlin, Berlin, Germany, 1990.

[23] M. KRENTEL, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.

[24] S. KURTZ, *On the random oracle hypothesis*, Inform. Control., 57 (1983), pp. 40–47.

[25] S. KURTZ, S. MAHANEY, AND J. ROYER, *The isomorphism conjecture fails relative to a random*

oracle, in Proc. 21st ACM Symposium on Theory of Computing, ACM Press, May 1989, pp. 157–166.

[26] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618–627.

[27] C. PAPADIMITRIOU AND M. YANNAKAKIS, *The complexity of facets (and some facets of complexity)*, J. Comput. System Sci., 28 (1984), pp. 244–259.

[28] C. PAPADIMITRIOU AND S. ZACHOS, *Two remarks on the power of counting*, in Proc. 6th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science 145, Springer-Verlag, Berlin, New York, 1983, pp. 269–276.

[29] U. SCHÖNING, *Probabilistic complexity classes and lowness*, in Proc. 2nd Structure in Complexity Theory Conference, IEEE Computer Society Press, June 1987, pp. 2–8.

[30] S. TODA, *On the computational power of* PP *and* $\oplus$P, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, October/November 1989, pp. 514–519.

SIAM J. COMPUT.
Vol. 20, No. 6, pp. 1157–1189, December 1991
© 1991 Society for Industrial and Applied Mathematics
011

AN ALMOST LINEAR-TIME ALGORITHM FOR THE DENSE SUBSET-SUM PROBLEM*

ZVI GALIL† AND ODED MARGALIT‡

Abstract. This paper describes a new approach for solving the subset-sum problem. It is useful for solving other NP-hard problems. The limits and potential of this approach are discussed. The approach yields an algorithm for solving the dense version of the subset-sum problem. It runs in time $O(\ell \log \ell)$, where ℓ is the bound on the size of the elements. But for dense enough inputs and target numbers near the middle sum, it runs in time $O(m)$, where m is the number of elements. Consequently, it improves the previously best algorithms by at least one order of magnitude and sometimes by two. The algorithm yields a characterization of the set of subset sums as a collection of arithmetic progressions with the same difference. This characterization is derived by elementary number-theoretic and algorithmic techniques. Such a characterization was first obtained by using analytic number theory and yielded inferior algorithms.

Key words. subset-sum, integer programming, dense problems, NP-hard

AMS(MOS) subject classifications. 05A05, 05A17, 10A45, 10L02, 10L20, 68C25

1. Introduction.

There are several ways to cope with the NP-hardness of optimization problems. One is to look for an approximate solution rather than the optimum. There is a vast literature about approximation algorithms and approximation schemes. For some problems there are very good approximation algorithms, for others, the problem is still NP-hard even if we settle for an approximate solution. Another way to cope with complexity is to settle for the average case or to allow probabilistic algorithms. There are cases where this approach has paid off and faster algorithms have been discovered. But this has not been the case with NP-hard optimization problems.

A third approach of coping with NP-hardness is to try to restrict the problem and design a polynomial-time algorithm. Here too, there have been mixed results. Some problems remain NP-hard even when restricted quite severely; others become feasible. In this paper we follow and extend the third approach. A restriction of the problem that allows a polynomial-time algorithm is known. However, the resulting algorithm has cubic time bound. We impose an additional restriction that allows much better algorithms and consequently much larger instances can be solved by these algorithms.

A novelty of our algorithm is the use of elementary number theory to design algorithms for solving an integer programming problem. It might be expected that this would be the natural tool for solving such problems. But we do not know of other such examples.

We use the following notations: given a set D of integers, $S_D = \sum_{a \in D} a$ is the sum of the elements of D and $D^* = \{S_E \mid E \subseteq D\}$ is the set of subset sums of D. The *subset-sum problem* is: Given a set A of m distinct integers in the interval $[1, \ell]$ and an integer N, find a subset $B \subseteq A$ such that $S_B \leq N$ and there is no $C \subseteq A$ such that $S_B < S_C \leq N$.

This problem is known to be NP-hard [14] but not in the strong sense: there is a pseudopolynomial algorithm for solving it [8]. A simple m stage dynamic program

* Received by the editors July 11, 1990; accepted for publication January 23, 1991.

† Department of Computer Science, Columbia University, New York, New York 10027; and Department of Computer Science, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel. The work of this author was partially supported by National Science Foundation grants CCR-8814977 and CCR-9014605.

‡ Department of Computer Science, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel.

solves the problem. The ith stage finds all subset sums of the ith prefix of A which are not larger than N. The time is $O(N)$ per stage, for a total time of $O(mN)$. The worst case is when N is $\Omega(\ell m)$; in this case the time is $\Theta(m^2\ell)$. The space needed by this algorithm can be as large as $\Theta(\ell m)$ just to find S_B, and $\Theta(m^2\ell)$ for finding B itself.

We derive an algorithm that is two orders of magnitude better than the dynamic programming approach, but works in a slightly more restricted domain. We impose two restrictions: we consider only *dense* instances, and we consider only an interval of target numbers around $S_A/2$. Note that the target numbers near the middle sum are the hardest cases for the dynamic programming algorithm. In our case $\ell > \ell_0$, $m^2 > c\ell \log^2 \ell$, and $L < N < S_A - L$ for $L << S_A$. Our algorithm runs in time $O(\ell \log \ell)$, but for dense enough instances and target numbers near the middle sum, its time bound is linear ($O(m)$).

The algorithm consists of a preprocessing stage, which is followed by a stage that depends on N and can be repeated for different target numbers. The preprocessing stage takes between $O(m)$ and $O(\ell \log \ell)$ time. The second stage (for a given N) takes only constant time to find S_B and $O(\log^2 m)$ time to obtain a characterization of B (only $\log m$ for dense enough inputs). Of course, we may need $\Theta(m)$ time to list the elements of B.

Our paper essentially concludes an interplay between analytic number theory and algorithm design. In the Appendix we sketch this history of "back and forth" between analytic number-theoretical methods and elementary ones. Freiman [10], using methods from analytical number theory, analyzed the structure of A^* for dense subset-sum problems. More recently, the structure was used to derive algorithms which improved the dynamic programming approach. The final algorithm derived using analytic number theory [7] requires $O(\ell^2 \log \ell)$ time. This bound is the same as for dynamic programming for the lowest density and as the density increases, its improvement increases. Our new algorithm is faster by at least one, sometimes two, orders of magnitude. Moreover, our algorithm yields an elementary proof of the characterization of A^* by constructing the sets whose sums are the corresponding elements of A^*.

In §2 we give a sketch of our algorithm. In §3 and §4 we describe the two major steps of the algorithm. In §3 we describe an efficient algorithm that constructs a set A^1 which has no small "almost-divisors" (this terminology is defined there) and prove an important property of this set. In §4 we describe an efficient constructive algorithm for finding a long arithmetic progression with small difference in the set of subset sums. This part is the heart of our algorithm. In §5 we fill in all the details of a weak version of the algorithm. In §6 we improve the performance of our algorithm: we speed it up and enlarge the interval $(L, S_A - L)$ in which it operates.

In §7 we investigate the limits of our approach. We prove that for lower density the characterization does not hold. We also show that several NP-hard problems cannot have an efficient solution even for the dense cases. In §8 we summarize our results and discuss open problems for future research.

2. A sketch of the algorithm. We describe the algorithm that computes B. A modification of the algorithm (obtained by deleting several steps) can compute only S_B. The algorithm consists of six steps. The first five are the preprocessing. The algorithm partitions the input set A into three parts: $A = A^1 \uplus A^2 \uplus A^3$. It first constructs A^1. A^2 consists of the μ smallest elements in $A \setminus A^1$ (μ will be defined later). A^3 contains the remaining elements: $A^3 \equiv A \setminus (A^1 \uplus A^2)$. We show that

$S_{A^3} \geq \frac{1}{2} S_A$. We now list the six steps of the algorithm.

1. In §3 we show how to reduce the problem to the case where A satisfies $A^* \,(\text{mod } d) = [0, d)$ for every small enough integer d.

$$A\,(\text{mod } d) \stackrel{\text{def}}{=} \{0 \leq i < d : \exists\, a \in A,\ a = i\,(\text{mod } d)\}.$$

   We then explain how to compute a not too large set A^1 satisfying

$$(A^1)^*\,(\text{mod } d) = [0, d)$$

   for every small enough integer d.
2. In §4 we generate a sequence of sets $B_i^2 \subseteq A^2$ for $0 \leq i < 2\ell$, such that $S_{B_i^2} = s + id_0$, for some integer s and a specific small integer d_0.
3. Using dynamic programming (modulo d_0) we construct for all $0 \leq i < d_0$, sets $B_i^1 \subseteq A^1$ such that $S_{B_i^1} = i\,(\text{mod } d_0)$ and $S_{B_i^1} \leq \ell d_0$.
4. We then join the output of the two previous steps to obtain for $0 \leq i < \ell$ the sets $B_i^{12} \subseteq A^1 \cup A^2$ such that $S_{B_i^{12}} = \tilde{L} + i$, where $\tilde{L} = s + \ell d_0 \leq L$.
5. We compute the list $\{\sigma_1, \sigma_2, \cdots, \sigma_{|A^3|}\}$ where σ_i is the sum of the first i elements in A^3. The elements of this list satisfy $\sigma_{i+1} - \sigma_i \leq \ell$ (since $\max A^3 \leq \ell$).

   Step 6 is performed for any given N. In our case (assuming $A^*\,(\text{mod } d) = [0, d)$) $N = S_B \in A^*$ (as will be shown).
6. If $N > S_A/2$ then we solve the complement problem: $N' = S_A - N$ and take $B = A \setminus B'$. To find B, we use binary search on the σ-list to find σ_j such that $N - \tilde{L} - \ell < \sigma_j \leq N - \tilde{L}$. Then,

$$B \equiv \{a_1^3, a_2^3, \cdots, a_j^3\} \cup B^{12}_{N - \sigma_j - \tilde{L}}$$

   satisfies $S_B = N$ and solves our problem.

In steps 3 and 4 we actually obtain a characterization of $\{B_i^1\}$ and $\{B_i^{12}\}$. This characterization enables us to compute the corresponding set if i is given. (Note that for a given N we compute only one such set for one particular i.) We compute a representation of B in $O(\log^2 m)$ time: the binary search for σ_i takes $O(\log m)$ and we get a representation of $B^{12}_{N-\sigma_i-\tilde{L}}$ in time $O(\log^2 m)$ (only $O(\log m)$ for dense enough inputs). The latter follows from the constructions of the following sections. The details that are omitted from this description will be given in §5.

We use the fact that $\ell > \ell_0$ in several cases throughout the paper without computing it (ℓ_0) precisely.

There is a small improvement that can be added to speed up the algorithm. The bottleneck of time complexity of this step, is building the arithmetic progression from the set A_2. To reduce the time complexity of this step we take a subinterval $\Lambda \subseteq [1, \ell]$ of length λ and choose A_2 from $\Lambda \cap A$. There is a trade-off between the time of the solution and the quality of the algorithm since on one hand we want to take λ as small as possible (to save time), but on the other hand we need λ to be large enough so that A_2 will contain enough elements to build the long arithmetic progression. These two contradicting constraints become harder to fulfill as the density of A decreases. This time improvement is described in §6.

3. A subset with no small almost-divisors. An integer d is an almost-divisor of a given set if almost all the elements of the set are divisible by d. The form of the theorem proved using analytic number theory is the following: If the set A has no small almost-divisors, then $(L, S_A - L) \subset A^*$; i.e., there is a complete interval of subset sums. Note that the condition that A has no small almost-divisors is necessary, because if all the elements are divisible by some d, so are all subset sums, and if almost all are divisible by d, it is to be expected that some residues modulo d are not in $A^* \pmod d$ and therefore some integers near the middle sum cannot be subset sums. For arbitrary A, this theorem is then transformed into the characterization of $A^* \cap (L, S_A - L)$ as a collection of arithmetic progressions with the same small difference.

The theorem mentioned above has required a proof that uses analytic number theory. As a result of our new algorithm, it now has an elementary proof which is quite complicated. In contrast, a similar-looking theorem can be very easily proved using elementary means: If the set A has no small almost-divisors, then $A^* \pmod d = [0, d)$ for all small enough d. (Note that this theorem follows from the theorem above, which we are not allowed to use.) This simple theorem is from Chaimovich [4] and for completeness is proved here as Lemma 3.3.

The algorithm computes all small almost-divisors of the given set A, and then it computes d_0, their least common multiple, which is also shown to be a small almost-divisor of A. This computation has to be done carefully to achieve the claimed time bound because the obvious way is far too inefficient. We build several sets of candidates for d_0, each set more accurate than its predecessor. First we examine only prime powers as candidates for d_0 and check them against a small subset of A. Then we slowly enlarge the subset and finally drop the prime power restriction.

If $d_0 = 1$ we then choose enough (but not too many) elements of A to form the set A^1: we choose enough nondivisors for every potential small divisor d. Thus A^1 has no small almost-divisors and therefore it satisfies $(A^1)^* \pmod d = [0, d)$ for all small enough d. The main algorithm uses this property for one specific d, the difference of the arithmetic progression obtained in the next section.

If $d_0 > 1$, then all but a small number of elements of A are divisible by d_0. Let $A(d_0)$ be those elements of A divisible by d_0. Applying dynamic programming modulo d_0 to $A \setminus A(d_0)$, we solve the subset-sum problem modulo d_0 obtaining $M \equiv S_B \pmod{d_0}$ (note that S_B is not known yet) and a set $C \subseteq A \setminus A(d_0)$ with $S_C = M$. It follows that S_B is the maximal number not larger than N satisfying $S_B \equiv M \pmod{d_0}$. Then we are left with the task of finding $D \subseteq A(d_0)$ with $S_D = S_B - M$ ($B = C \cup D$). Dividing all these elements by d_0, we observe that $A(d_0)/d_0$ has no small almost-divisors; i.e., the corresponding d_0 is 1. (This needs a proof.) Thus, the remaining task is solving the subset-sum problem for a set with no small almost-divisors. This explains why we may assume without loss of generality that the given set A has no small almost-divisors.

Note that in the case that $d_0 = 1$ we obtain an interval of subset sums, and if $d_0 > 1$ we obtain a collection of arithmetic progressions with difference d_0. Each arithmetic progression corresponds to an element of $A^* \pmod{d_0}$. This indicates how the characterization is obtained from our algorithm and how the theorem about the interval of subset sums is transformed into a characterization of A^* near $S_A/2$ as a collection of arithmetic progressions.

Notation.

- $X_{(r)}$ stands for the first smallest r elements of X.

- $X(i,r) \stackrel{\text{def}}{=} \{x \in X : x \equiv i \pmod{r}\}$.
- $d \wr_r X$ (read as d r-almost-divides X), if and only if $|X \setminus X(0,d)| < r$.
- p denotes a prime number and q a prime power.

Now we prove some simple properties of the "almost-divides" relation.

LEMMA 3.1. *Let X be a set of different integers and d and r two integers such that $d \wr_r X$. Then $d < \max(X)/(|X| - r)$.*

Proof. If $d \wr_r X$ then $|X(0,d)| > |X| - r$, but there are only $\lfloor \max(X)/d \rfloor$ different integers divisible by d in the interval $[1, \max(X)]$ which contains X, so

$$\frac{\max(X)}{d} \geq \left\lfloor \frac{\max(X)}{d} \right\rfloor \geq |X(0,d)| > |X| - r. \qquad \square$$

LEMMA 3.2. *Let X be a set of different integers and x, y, and τ be integers such that $x, y < \tau$, $x \wr_x X$, $y \wr_y X$, and $\tau(|X| - 2\tau) \geq \max(X)$; then $z < \tau$ and $z \wr_z X$ where $z = \operatorname{lcm}(x,y)$.*

Proof. If $z \in \{x, y\}$, the lemma obviously holds. So assume $z \notin \{x, y\}$ and thus $z \geq x + y$. By definition, $X(0,z) = X(0,x) \cap X(0,y)$, so

$$|X \setminus X(0,z)| \leq |X \setminus X(0,x)| + |X \setminus X(0,y)| \leq x - 1 + y - 1 < z.$$

So $z \wr_z X$. Since $z \wr_{x+y-1} X$, from Lemma 3.1,

$$z < \frac{\max(X)}{|X| - (x + y - 1)} < \frac{\max(X)}{|X| - 2\tau} \leq \tau. \qquad \square$$

LEMMA 3.3. *Let t be an integer and X be a set of integers such that*

$$\forall\ 1 < d < t, \qquad \neg d \wr_d X;$$

then for all $d < t$, $X^* \pmod{d} = [0, d)$.

Proof. We prove that $X^* \pmod{d} = [0,d)$ by induction on d. The induction base ($d = 1$) is trivial. The induction step is: Let $x_1, \cdots, x_r$, $r \geq d$ be elements of X which are not divisible by d. Define a sequence of sets:

$$C_0 = \emptyset, \qquad C_i = C_{i-1} + \{0, x_i\} \pmod{d}.$$

If $C_i \neq C_{i-1}$ for $1 \leq i \leq d$, then $|C_d| \geq d$ and the proof is complete. Otherwise, for some $1 \leq i \leq d$, we have $C_i = C_{i-1}$, which implies that

$$\text{if } c \in C_i \qquad \text{then } c + x_i \pmod{d} \in C_i.$$

Applying the above k times we get that

$$\text{if } c \in C_i \qquad \text{then } \forall\ k,\ c + kx_i \pmod{d} \in C_i.$$

Let $d' \stackrel{\text{def}}{=} \gcd(x_i, d)$. Since $d' < d$, by induction assumption, $X^* \pmod{d'} = [0, d')$.

Assume $x = rd'$. Now x_i/d' has an inverse modulo d, so there is an integer u such that $ux_i/d' \equiv 1 \pmod{d}$ and therefore $ux_i \equiv d' \pmod{d}$, and there is an integer r such that $rx_i \equiv x \pmod{d}$. Hence if $c \in C_i$, then $c + x \pmod{d} \in C_i$. Since

$$X^* \equiv C_r \equiv B + C_i \pmod{d}\ (B \stackrel{\text{def}}{=} \{x_{i+1}, \cdots, x_k\}^*),$$

we also have that if $c \in X^* \pmod{d}$, then $c + x \in X^* \pmod{d}$.

We showed that all residues modulo d' are in X^* and that adding a multiple of d' preserves membership in X^* (mod d). Therefore, all residues modulo d are in X^* (mod d). □

The main theorem of this section follows.

THEOREM 3.4. *Given a set of integers* $A = \{1 \leq a_1 < a_2 < \cdots < a_m \leq \ell\}$ *and an integer* t *such that*

$$\frac{5\ell}{m} < t < \frac{m}{30 \log_2 \ell}, \tag{1}$$

we can find, in $O(m + t^2)$ *time, an integer* $d_0 < t$ *such that*

$$d_0 \wr_{d_0} A, \tag{2}$$

$$\forall\, d < t, \qquad \neg d \wr_d \frac{A(0, d_0)}{d_0}, \tag{3}$$

and

$$S_{A \setminus A(0,d_0)} < \frac{\ell}{\binom{m-t}{2}} S_A. \tag{4}$$

If $d_0 = 1$, *we find, in the same time, a subset* $\Delta \subset A$ *such that*

$$\forall\, d < t, \qquad \neg d \wr_d \Delta, \tag{5}$$

$$S_\Delta < \frac{1}{3} S_A, \tag{6}$$

and

$$|\Delta| < \frac{1}{3}|A|. \tag{7}$$

Note that condition (1) is not vacuous because of the density assumption $m^2 > 150\ell \log_2 \ell$. We will use the theorem as follows. First apply it to the original set. Then if $d_0 > 1$, by part (2) d_0 almost-divides A and by part (3) $A(0, d_0)/d_0$ has no almost-divisors smaller than t. In this case we apply the theorem again to the set $A(0, d_0)/d_0$, so without loss of generality $d_0 = 1$ and we can find the relatively small subset of A which has no almost-divisors smaller than t.

Our proof is constructive: we design an algorithm for computing d_0 and Δ and prove its correctness. We show that d_0 is the maximal element in $[1, t)$ which d-almost divides A. Computing d_0 in the straightforward way takes too much time, since we should check $|A|$ divisibility tests for each one of the t candidates for d_0. So we compute d_0 in several steps. We use the fact that the almost-divide relation is monotone in the following sense:

$$\text{if } X \supseteq Y \quad \text{and} \quad d \wr_r X \qquad \text{then } d \wr_r Y.$$

We build several sets of candidates for d_0. Each set is based on more checks and thus is more accurate. At first we examine only prime powers as candidates for d_0 and check them against a small subset of A. Then we slowly enlarge the subset of A, and finally we drop the prime powers restriction on d_0.

THE ALGORITHM

Recall that q stands for a prime power.

1. Compute $B_1 \stackrel{\text{def}}{=} \{1 < q < t : q \wr_t A_{(2t)}\}$.
   Denote $b \stackrel{\text{def}}{=} 3t \log_2 \ell$.
2. Compute $B_2 \stackrel{\text{def}}{=} \{1 < q < t : q \wr_t A_{(b)}\}$.
3. Compute $B_3 \stackrel{\text{def}}{=} \{1 < q < t : q \wr_t A_{(m/4)}\}$.
4. Compute $B_4 \stackrel{\text{def}}{=} \{1 < d < t : d \wr_t A_{(m/4)}\}$.
5. Compute $B_5 \stackrel{\text{def}}{=} \{1 < d < t : d \wr_d A\}$.
   Denote $d_0 \stackrel{\text{def}}{=} \max(B_5)$, $d_0' \stackrel{\text{def}}{=} \max(B_4)$. Apply step 6 if $d_0 = 1$.
6. Build $\Delta \stackrel{\text{def}}{=} A_{(m/4)} \cup \bigcup_{d \in B_4} C_d$ where C_d are d elements from $A \setminus A(0, d)$.

It is easy to see that $B_1 \supseteq B_2 \supseteq B_3$ and $B_4 \supseteq B_5$.

LEMMA 3.5. $|B_1| < 2 \log_2 \ell$.

Proof. Any integer $n = \prod_i p_i^{k_i}$ has $\sum_i k_i$ divisors which are prime powers and

$$n = \prod_i p_i^{k_i} \geq \prod_i 2^{k_i} = 2^{\left(\sum_i k_i\right)}.$$

Therefore, each $a \in A$ cannot be divisible by more than $\log_2 \ell$ prime powers. Denote

$$\delta(i, q) \stackrel{\text{def}}{=} \begin{cases} 1, & q | a_i, \\ 0, & \text{otherwise.} \end{cases}$$

Counting the divisors in two ways we get:

$$2t \log_2 \ell \geq \sum_{i=1}^{2t} \sum_q \delta(i, q) = \sum_q \sum_{i=1}^{2t} \delta(i, q)$$

and

$$|B_1| = \left| \left\{ q : \sum_{i=1}^{2t} \delta(i, q) > t \right\} \right| < 2 \log_2 \ell. \qquad \square$$

LEMMA 3.6. $\mathrm{lcm}(B_2) < \ell / (t \log \ell)$.

Proof. Clearly $A_{(b)}(0, \mathrm{lcm}(B_2)) = \cap_{q \in B_2} A_{(b)}(0, q)$ so

$$|A_{(b)} \setminus A_{(b)}(0, \mathrm{lcm}(B_2))| = |\cup_{q \in B_2} (A_{(b)} \setminus A_{(b)}(0, q))| \leq |B_2| \cdot t < 2t \log_2 \ell.$$

Therefore

$$\mathrm{lcm}(B_2) \wr_{2t \log_2 \ell} A_{(b)}$$

and by Lemma 3.1,

$$\mathrm{lcm}(B_2) < \frac{\ell}{t \log_2 \ell}. \qquad \square$$

LEMMA 3.7. $\mathrm{lcm}(B_3) = O(\ell / m)$.

Proof. Since $B_3 \subseteq B_2$, we get, as in the previous lemma,

$$\mathrm{lcm}(B_3) \wr_{2t \log_2 \ell} A_{(m/4)}.$$

By Lemma 3.1 ($2t\log_2 \ell < m/15$)

$$\operatorname{lcm}(B_3) < \frac{\ell}{m/4 - m/15}. \qquad \square$$

LEMMA 3.8. *$d_0' = \operatorname{lcm}(B_3) = \operatorname{lcm}(B_4) < t$ and $d_0 = \operatorname{lcm}(B_5) < t$.*

Proof. Clearly $B_3 \subseteq B_4$, so $\operatorname{lcm}(B_3)|\operatorname{lcm}(B_4)$. Let $d \in B_4$ and $d = \Pi_{i=1}^{n} p_i^{k_i}$ be the prime factorization of d. $d \wr_t A_{(m/4)}$ implies

$$p_i^{k_i} \wr_t A_{(m/4)},$$

so $p_i^{k_i} \in B_3$ for all $1 \le i \le n$ and hence $d|\operatorname{lcm}(B_3)$, which proves that $\operatorname{lcm}(B_4)|\operatorname{lcm}(B_3)$ and thus $\operatorname{lcm}(B_3) = \operatorname{lcm}(B_4)$.

From condition (1), $t(m/4 - 2t) \ge \ell$. So we can use Lemma 3.2 and get that B_4 is closed under lcm; therefore $\operatorname{lcm}(B_4) \in B_4$. Similarly $t(m - 2t) \ge \ell$ implies $\operatorname{lcm}(B_5) \in B_5$. $\square$

Part (2) of Theorem 3.4 follows from Lemma 3.8. The following lemma shows that part (3) of Theorem 3.4 holds.

LEMMA 3.9. *For all $1 < d < t$, $\neg d \wr_d A(0, d_0)/d_0$.*

Proof. Denote $X \stackrel{\text{def}}{=} A(0, d_0)/d_0$. Assume $d < t$ and $d \wr_d X$. By definition, $X(0, d) = A(0, dd_0)/d_0$, so $dd_0 \wr_{d+d_0-1} A$. From Lemma 3.1 we have

$$dd_0 < \frac{\ell}{m - d - d_0 + 1} < \frac{\ell}{m - 2t} \le t.$$

and from the definition of d_0 we get $dd_0 \le d_0$, so $d = 1$. $\square$

LEMMA 3.10.

$$S_{A \setminus A(0,d_0)} < \frac{\ell}{\binom{m - d_0}{2}} S_A.$$

Proof. From (2) it follows that

$$|A \setminus A(0, d_0)| < d_0,$$

so

$$S_{A \setminus A(0,d_0)} < \ell d_0.$$

On the other hand,

$$S_A \ge S_{A(0,d_0)} > d_0 \binom{m - d_0}{2}.$$

Combining these two inequalities yields the desired result. $\square$

The last lemma proved (4) since $d_0 < t$. The rest of the proof is for the case $d_0 = 1$. The construction of C_d in step 6 is valid, because there are d nondivisors for each $d < t$. The following lemma shows that (5) of Theorem 3.4 holds for the set Δ constructed in step 6.

LEMMA 3.11. *For all $1 < d < t$, $\neg d \wr_d \Delta$.*

Proof. If $d \in B_4$, then $\neg d \wr_d C_d$ and thus $\neg d \wr_d \Delta$; otherwise $\neg d \wr_t A_{(m/4)}$ and thus $\neg d \wr_d \Delta$. $\square$

Lemma 3.12 proves (7).

LEMMA 3.12. $|\Delta| < \frac{1}{3}|A|$.

Proof. Obviously, $B_4 \subseteq \{d : d|d'_0\}$. Denote $C(d'_0) \stackrel{\text{def}}{=} \bigcup_{d|d'_0} C_d$, so

$$|\Delta| \le |A_{(m/4)} \cup C(d'_0)| \le \frac{m}{4} + \sum_{d|d'_0} d.$$

Using the approximation

$$\sum_{d|d'_0} d \le \sum_{i=1}^{d'_0} \frac{d'_0}{i} < d'_0 \left(1 + \int_1^{d'_0} \frac{1}{x}\,dx\right) \le d'_0 \log(ed'_0), \tag{8}$$

we get:

$$|\Delta| < \frac{m}{4} + d'_0 \log(ed'_0)$$

and from Lemma 3.8, $d'_0 < t < m/(30 \log_2 \ell)$ and

$$|\Delta| < \frac{m}{4} + \frac{m}{30 \log_2 \ell} \left(\log m - \log\left(\frac{30}{e} \log_2 \ell\right)\right) < \frac{m}{3}. \qquad \square$$

LEMMA 3.13. $S_A > m^2 d'_0/8.01$.

Proof. $A_{(m/4)}$ has more than $(m/4) - d'_0$ different elements which are divisible by d'_0. Therefore

$$S_{A_{(m/4)}} > \binom{\frac{m}{4} - d'_0}{2} d'_0.$$

Using the fact that $d'_0 < t$ (d'_0 is an order of magnitude less than m and we are using here the fact that $\ell > \ell_0$) and $S_A \ge 4S_{A_{(m/4)}}$ we get that $S_A > (m^2/8.01)d'_0$. $\square$

LEMMA 3.14. $S_\Delta < \frac{1}{3}S_A$.

Proof. $\Delta \subseteq A_{(m/4)} \cup C(d'_0)$. We bound the sum in parts. The first part obviously satisfies

$$S_{A_{(m/4)}} < \frac{1}{4} S_A.$$

We bound the second part by approximating the sum as the product of the cardinality and the maximal number:

$$S_{C(d'_0)} < \ell d'_0 \log(ed'_0).$$

Using the previous lemma and the density condition implied by condition (1) we get:

$$S_{C(d'_0)} < \frac{8.01 S_A}{m^2} 1.5 \ell \log_2 \ell < S_A \frac{8.01 \cdot 1.5}{150} < \frac{1}{12} S_A.$$

Combining the two parts we get:

$$S_\Delta < S_A \left(\frac{1}{4} + \frac{1}{12}\right) = \frac{1}{3} S_A. \qquad \square$$

The last lemma proved (6). To complete the proof we analyze the time complexity by elaborating some algorithmic details. Note that computing *lcm* of a set of n elements can be done in $O(n \log n)$ operations ($O(\log n)$ per lcm computation). Computing the divisors of a given integer d is even simpler: it takes $O(d)$ time. These costs are negligible because $d, n < t$.

1. $O(t^2)$: There are less than t candidates for B_1 and for each one of them we check $2t$ divisibility conditions.
2. $O(t\log^2 \ell)$: There are $b = 3t\log_2 \ell$ divisibility checks for each one of the $|B_1| < 2\log_2 \ell$ candidates. (Because $B_2 \subseteq B_1$ and from Lemma 3.5.)
3. $O(m + (\ell/t))$: First compute, in $O(m)$ time, $|A(i, \mathrm{lcm}(B_2))|$ for all i in the interval $[0, \mathrm{lcm}(B_2))$; this can be done by taking all the elements of A one by one and computing the remainder modulo $\mathrm{lcm}(B_2)$. Checking all the elements of B_2 as candidates for B_3 takes $O(\mathrm{lcm}(B_2))$ per element since

$$\text{for } q \in B_2, \qquad |A(0,q)| = \sum_{i=0}^{\mathrm{lcm}(B_2)/q} |A(iq, \mathrm{lcm}(B_2))|.$$

   The time complexity follows from Lemma 3.6 ($\mathrm{lcm}(B_2) = O(\ell/t\log \ell)$) and from Lemma 3.5, (since $|B_2| \le |B_1| < 2\log_2 \ell$).
4. $O(m + t\ell/m)$: As in the previous step we compute, in $O(m)$ time,

$$\{|A(i, \mathrm{lcm}(B_3))|\}_{i=0}^{\mathrm{lcm}(B_3)-1}$$

   and then use them to check all t candidates for B_4 in $\mathrm{lcm}(B_3) = O(\ell/m)$ divisions each (Lemma 3.7).
5. $O(m + t\ell/m)$: as in step 4.
6. $O(m + t^2)$: This step consists of two substeps. The first substep computes $A(i, d_0')$ for $i \in (0, d_0')$. The second builds the sets C_d. The first substep takes $O(m)$ time, as before. The second substep fills up the sets C_d in the obvious way:

   for $d \in \{\text{divisors of } d_0'\}$ do $C_d \leftarrow \emptyset$.
   for i:=1 to $d_0' - 1$ do
      for $d \in \{\text{divisors of } d_0'\}$ do
         if ($d \not\equiv 0 \pmod i$) and $|C_d| < d$) then
            $C_d \leftarrow C_d \cup A(i, d_0')_{(\min\{d - |C_d|, |A(i, d_0')|\})}$.

   The second substep takes $O(d_0'^2) = O(t^2)$ time.

The total complexity is:

$$O\left(t^2 + t\log^2 \ell + m + \frac{\ell}{t} + t\frac{\ell}{m}\right).$$

If $t > \log^2 \ell$ then $t\log^2 \ell < t^2$, otherwise $t\log^2 \ell < m$ (here we use again the fact that $\ell > \ell_0$), so that in any case $t\log^2 \ell = O(m + t^2)$, $\ell/t = O(m)$, and $t\ell/m = O(t^2)$, which makes the total complexity $O(m + t^2)$.

4. Constructing an arithmetic progression. In this section we describe an efficient algorithm for finding an arithmetic progression; more precisely, we constructively prove the following theorem.

THEOREM 4.1. *Given a set Φ of μ distinct integers in an interval of length λ and an integer $\ell \ge \lambda$, where*

$$\mu^2 > 50\lambda\log_2^2 \lambda + 200\ell\log_2 \lambda, \tag{9}$$

we can find in $O(\lambda\log\lambda + \mu\log^2\lambda)$ time a characterization of subsets $\{B_i\}_{i=1}^{2\ell} \subseteq \Phi$ such that $S_{B_i} = s_0 + ig_r$ for some integers s_0 and g_r. This characterization can be translated into an elements list in time proportional to the cardinality of the set.

There is a simple way to generate several short arithmetic progressions: For $1 \le i \le \lambda$, consider the sets P_i of pairs, $P_i \equiv \{(a,b) \in \Phi^2 \mid a-b=i\}$. There are $\Omega(\mu^2)$ pairs (a,b) and thus (by the pigeon-hole argument) there are many pairs with the same difference. We first take enough of the largest P_i's. The pairs in each P_i are disjoint, but pairs in different P_i's may intersect. We clean these P_i's by deleting pairs to restore the disjointness property (see step 4 in the algorithm below). The new sets P_i' are still large enough. Now, we take from each pair in P_ρ' and P_σ' one of its elements, either the large one or the small one. Taking $k-i$ large elements from P_ρ' and i large elements from P_σ' gives us a sequence of sets $D_i \subset \Phi$ with $S_{D_i} = \xi + k\rho + i(\sigma - \rho)$ — an arithmetic progression. (ξ is the sum of the small elements in P_ρ' and P_σ'.)

The arithmetic progression generated this way is far too short, but we can generate many of them and then combine them in two different ways in order to generate longer and longer arithmetic progressions. We start with the progression of minimal difference. We inductively combine the $(i+1)$st progression to the combined arithmetic progression of the first i progressions. Our construction guarantees that the difference of the $(i+1)$st arithmetic progression is a multiple of the difference of the ith combined arithmetic progression and the length of the i combined arithmetic progression should be at least as long as the difference of the $(i+1)$st progression. This join process is illustrated in Fig. 1. An element of the combined progression is the sum of an appropriate element from each of the two progressions.

We guarantee that the difference in the $(j+1)$st step will be a multiple of the difference of the jth step by taking σ and ρ from the same residue class modulo the previous difference. Actually we choose them from the same residue class modulo twice the previous difference so that the difference will be strictly increasing. We choose them from the largest residue class, to make the arithmetic progression as long as possible, and restrict ourselves to this residue class to save time. More details are given in step 3 of the algorithm below.

The resulting arithmetic progression has a small difference and is sufficiently long for high densities ($\mu > c(\ell \log \ell)^{2/3}$). For smaller densities we repeat the process above a number of times and combine the resulting progressions. Now we do not have the divisibility property. We combine two arithmetic progressions with differences d_1 and d_2 and length of at least $\mathrm{lcm}(d_1, d_2)$ to generate an arithmetic progression with difference $\gcd(d_1, d_2)$ as shown in Fig. 2. Here too, an element of the combined progression is the sum of an appropriate element from each of the two progressions.

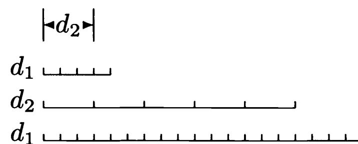


FIG. 1. *Combining when d_1 divides d_2.*

Notation.

- For $1 \le i \le \lambda$, $P_i \stackrel{\text{def}}{=} \{(a,b) \in \Phi^2 \mid a-b=i\}$ and $p_i \stackrel{\text{def}}{=} |P_i|$.
- $M \stackrel{\text{def}}{=} (1/2 \log(e\lambda)) \binom{\mu}{2}$.

For the set I defined below,

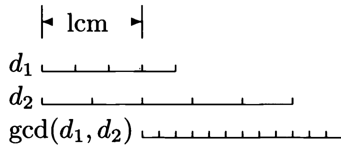


FIG. 2. *Combining when d_1 does not divide d_2.*

- $\alpha_d \stackrel{\text{def}}{=} |\{r : I(r,d) \neq \emptyset\}|$ (recall that $I(r,d) \stackrel{\text{def}}{=} \{i \in I : i \equiv r \pmod{d}\}$).
- Let $\{s + id\}_{i=0}^{n}$ be an arithmetic progression. We define the *length* of the progression as nd.

The algorithm below computes certain sets R_j^i, S_j^i. Each such set is contained in a set P_s for some s. All these sets are disjoint. These sets will be used as building blocks for generating arithmetic progressions, which will be combined to produce the desired arithmetic progressions. In step 1 we compute the P_i's using the FFT to multiply polynomials. In step 2 we compute a subset I of the largest p_i's satisfying a certain condition. Step 2 is justified by Lemma 4.2. Step 3 computes pairs of elements of I, $\{\rho_j^i, \sigma_j^i\}_{i=0}^{r}{}_{j=1}^{k_i}$, in a double loop. They are used later to construct arithmetic progression of difference d_j^i, as indicated above, and proved in Lemma 4.6. In the inner loop we combine k_i arithmetic progressions together as shown in Fig. 1. The progressions are combined one by one, keeping the intermediate arithmetic progression with the same difference d_0^i and making it longer and longer. In the outer loop we combine the output of the inner loop in the same accumulative manner to a single arithmetic progression. The combining process is shown in Fig. 2. This time we make the difference smaller (g_i) while keeping the arithmetic progression long. Step 4 computes the disjoint sets

$$R_j^i \subseteq P_{\rho_j^i} \quad \text{and} \quad S_j^i \subseteq P_{\sigma_j^i}$$

of a predetermined size (β_j^i). The obvious way to compute these sets is too costly. A more efficient way is the following: generate the sets R_j^i and S_j^i one by one; each such set is chosen from the elements of Φ which were not used before. We compute each set $P_{\rho_j^i}$ in $O(m)$ time. Lemma 4.8 shows that this process works correctly.

THE ALGORITHM

1. Compute $\{p_i\}_{i=1}^{\lambda}$ by using the Fast Fourier Transform and the following equation

$$\sum_{\varphi\in\Phi} x^{\varphi} \cdot \sum_{\varphi\in\Phi} x^{-\varphi} = \sum_{i=1}^{\lambda}(-p_i)x^{-i} + |\Phi| + \sum_{i=1}^{\lambda} p_i x^i.$$

2. Compute I in the following way:
   Sort $\{p_i\}_{i=1}^{\lambda}$: $\{p_{j_1} \geq p_{j_2} \geq \cdots p_{j_\lambda}\}$.
   Let s be the minimal integer such that $sp_{j_s} > M$,
   then $I \stackrel{\text{def}}{=} \{j_1, j_2, \cdots, j_s\}$.
3. Compute $\{(\rho_j^i, \sigma_j^i)\}_{i=0}^{r}{}_{j=1}^{k_i}$ as follows:
   $g_0 \stackrel{\text{def}}{=} 0$. (Remember that $\gcd(0,i) = i$ and $\neg 0 | i$ for all $i \geq 1$.)
   $i \leftarrow 0$.

While ($|\{s : I(s, g_i) \neq \emptyset\}| > 1$) do
    begin
        $i \leftarrow i + 1$.
        Choose $\sigma_0^i,\ \rho_0^i \in I$ such that $\sigma_0^i > \rho_0^i,\ \sigma_0^i \not\equiv \rho_0^i \pmod{g_i}$, and
                $\sigma_0^i - \rho_0^i$ is minimal under these constraints.
        $j \leftarrow 0$.
        $d_0^i \stackrel{\text{def}}{=} \sigma_0^i - \rho_0^i$.
        $\delta \leftarrow 1; \qquad \delta_0 \leftarrow 0$.
        While ($\exists\ s,\ s = \delta_0 \pmod{\delta},\ |I(s, 2d_j^i)| > 1$) do
            begin
                Choose $0 \leq s_j^i < 2d_j^i$ such that $s_j^i \equiv \delta_0 \pmod{\delta}$ and
                        $|I(s_j^i, 2d_j^i)|$ is maximal under these constraints.
                $\delta \leftarrow 2d_j^i; \qquad \delta_0 \leftarrow s_j^i$.
                $j \leftarrow j + 1$.
                Choose $\sigma_j^i,\ \rho_j^i \in I(s_{j-1}^i, 2d_{j-1}^i)$ such that $\sigma_j^i > \rho_j^i$, and
                        $\sigma_j^i - \rho_j^i$ is minimal.
                $d_j^i \stackrel{\text{def}}{=} \sigma_j^i - \rho_j^i$.
            end
        $k_i \leftarrow j$.
        $g_i \stackrel{\text{def}}{=} \gcd(g_{i-1}, d_0^i)$.
    end
$r \leftarrow i$.

4. Compute subsets $R_j^i \subseteq P_{\rho_j^i},\ S_j^i \subseteq P_{\sigma_j^i}$ as follows:

Denote $$\beta_j^i = \begin{cases} \left\lceil \frac{8\ell}{|I|} \right\rceil & i = r, \quad j = k_i, \\ \left\lceil \frac{4\lambda}{|I|} \right\rceil & i = r - 1, \quad j = k_i, \\ 1 + \left\lceil \frac{2\lambda}{|I|} \right\rceil & j = k_i, \quad i < r - 1, \\ 1 + (d_{j+1}^i / d_j^i) & \text{otherwise.} \end{cases}$$

Sort $\Phi = \{\varphi_1, \varphi_2, \cdots, \varphi_\mu\}$; mark all the elements of Φ as unused.
For $i = 1$ to r do
    For $j = 0$ to k_i do
        begin
            $R_j^i \leftarrow \emptyset$.
            $i_1 \leftarrow 1$.
            $i_2 \leftarrow \mu$.
            While ($|R_j^i| < \beta_j^i$) do
                begin
                    While ($\varphi_{i_1} + \varphi_{i_2} > \rho_j^i$) do
                        $i_2 \leftarrow i_2 - 1$.
                    If ($\varphi_{i_1} + \varphi_{i_2} = \rho_j^i$ and both φ's are marked as unused)
                    then begin
                            $R_j^i \leftarrow R_j^i \cup \{(\varphi_{i_1}, \varphi_{i_2})\}$.
                            Mark both φ's as used.
                        end
                    $i_1 \leftarrow i_1 + 1$.
                end
            $S_j^i \leftarrow \emptyset$.
            $i_1 \leftarrow 1$.

```
        i_2 ← μ.
        While (|S_j^i| < β_j^i) do
            begin
                While (φ_{i_1} + φ_{i_2} > σ_j^i) do
                    i_2 ← i_2 − 1.
                If (φ_{i_1} + φ_{i_2} = σ_j^i and both φ's are marked as unused) then
                  begin
                    S_j^i ← S_j^i ∪ {(φ_{i_1}, φ_{i_2})}.
                    Mark both φ's as used.
                  end
                i_1 ← i_1 + 1.
            end
    end
```

Correctness proof.

LEMMA 4.2. *Step 2 of the algorithm is valid (i.e., s exists) and the resulting set I satisfies:*

$$\min_{i\in I} p_i > \frac{1}{2\lambda}\binom{\mu}{2}. \tag{10}$$

$$|I|\min_{i\in I} p_i > M. \tag{11}$$

$$|I| > \frac{M}{\mu}. \tag{12}$$

Proof. If s does not exists, then for all $1 \le \tau \le \lambda$, $\tau p_{i_\tau} \le M$, so by the definition of M,

$$\binom{\mu}{2} = \sum_{\tau=1}^{\lambda} p_{i_\tau} \le \sum_{\tau=1}^{\lambda} M\frac{1}{\tau} < \frac{1}{2}\binom{\mu}{2}. \tag{13}$$

This contradiction proves that s exists and therefore part (11) holds. Part (12) follows from part (11):

$$\min_{i\in I} p_i \le \max_{i\in I} p_i \le \mu$$

so

$$|I| \ge \frac{|I|\min_{i\in I} p_i}{\mu} > \frac{M}{\mu}.$$

Using a computation similar to (13) yields $\sum_{\tau=1}^{s-1} p_{i_\tau} < \frac{1}{2}\binom{\mu}{2}$. Hence p_{i_s} is chosen from more than $\frac{1}{2}\binom{\mu}{2}$ remaining pairs, which proves (pigeon-hole argument) that

$$p_{i_s} > \frac{1}{2\lambda}\binom{\mu}{2},$$

so (10) holds. □

LEMMA 4.3. $\alpha_{ab} \leq a\alpha_b$.

Proof. The proof is obvious. □

LEMMA 4.4. *d^i_j satisfy*

$$d^1_0 \leq \frac{\lambda}{|I|}. \tag{14}$$

For all $1 \leq i \leq r$ and $1 \leq j \leq k_i$,

$$2 \leq \frac{d^i_j}{d^i_{j-1}} \leq \frac{2\lambda}{|I|}. \tag{15}$$

and

$$\forall\, 1 \leq i \leq r, \qquad d^i_{k_i} \geq \frac{|I|}{2}\frac{d^i_0}{\alpha_{d^i_0}}. \tag{16}$$

Proof. Obviously, by (12), the definition of M, and (9), $|I| > 1$. By the pigeon-hole argument there exist two elements in I whose distance is not larger than $\lambda/|I|$, thus (14) holds. Using the same argument again (I has $\alpha_{2d^i_0}$ congruent classes modulo $2d^i_0$) and Lemma 4.3 we get that

$$|I(s^i_0, 2d^i_0)| \geq \frac{|I|}{\alpha_{2d^i_0}} \geq \frac{|I|d^i_0}{2d^i_0\alpha_{d^i_0}}.$$

This proves the base hypothesis ($j = 0$) of

$$|I(s^i_j, 2d^i_j)| \geq \frac{|I|}{2d^i_j}\frac{d^i_0}{\alpha_{d^i_0}}. \tag{17}$$

The induction step is proved by another pigeon-hole argument: $I(s^i_{j-1}, 2d^i_{j-1})$ has $2d^i_j/2d^i_{j-1}$ classes modulo $2d^i_j$ so

$$|I(s^i_j, 2d^i_j)| \geq \frac{|I(s^i_{j-1}, 2d^i_{j-1})|}{2d^i_j/2d^i_{j-1}} \geq \frac{|I|}{2d^i_j}\frac{d^i_0}{\alpha_{d^i_0}}.$$

Thus, as long as

$$d^i_{j-1} < \frac{|I|}{2}\frac{d^i_0}{\alpha_{d^i_0}}$$

we can iterate at least one more iteration (since $\left|I(s^i_j, 2d^i_j)\right| > 1$) and find d^i_j. This proves (16). The first half of (15) is immediate, since by the definition of d^i_j, for $j > 1$,

$$2d^i_{j-1} | d^i_j.$$

Using the pigeon-hole argument again we get σ^i_j and ρ^i_j in $I(s^i_{j-1}, 2d^i_{j-1})$ such that

$$d^i_j \stackrel{\text{def}}{=} \sigma^i_j - \rho^i_j \leq \frac{\lambda}{|I(s^i_{j-1}, 2d^i_{j-1})|},$$

and by using (17) it follows that

$$\frac{d^i_j}{d^i_{j-1}} \leq \frac{2\lambda\alpha_{d^i_0}}{|I|d^i_0} \leq \frac{2\lambda}{|I|}.$$

This completes the proof of the lemma. □

LEMMA 4.5. $d_0^{i+1} \leq \lambda/\alpha_{g_i}$.

Proof. d_0^{i+1} is chosen as the minimal difference in I not divisible by g_i. Therefore we can partition I into s subsets as follows. Sort the set I:

$$I = \{i_1 < i_2 < \cdots < i_{|I|}\}$$

and define

$$j_0 = 1, \quad j_{k+1} = \min_j\{i_j \neq i_{j_k} \pmod{g_i}\}, \quad j_s = |I| + 1.$$

Each "interval" $i_{j_{k-1}}, \cdots, i_{j_k - 1}$ has the same remainder modulo g_i

$$\neg g_i | i_{j_k} - i_{j_{k-1}},$$

and d_0^{i+1} is the minimal distance satisfying this property, hence $i_{j_k} - i_{j_{k-1}} \geq d_0^{i+1}$. From

$$\lambda \geq i_{j_s} - i_{j_0} \geq \sum_{k=1}^{s}(i_{j_k} - i_{j_{k-1}}) \geq s d_0^{i+1}$$

we get $d_0^{i+1} \leq \lambda/s$. Each "interval" has the same reminder modulo g_i, therefore $\alpha_{g_i} \leq s$; hence $d_0^{i+1} \leq \lambda/\alpha_{g_i}$. □

LEMMA 4.6. *Let R, S be subsets of Φ such that $R \subseteq P_\rho$ and $S \subseteq P_\sigma$; then $(R \cup S)^*$ contains an arithmetic progression with difference $\sigma - \rho$ and with $k = \min\{|R|, |S|\}$ elements. (Recall that $X^* \stackrel{\text{def}}{=} \{S_Y : Y \subseteq X\}$.) One can generate in constant time a description of the arithmetic progression from which an elements list can be built in time proportional to the set cardinality.*

Proof. Recall that R and S contains pairs of elements with the same difference. Denote by $R_{\{i\}}(S_{\{i\}})$ the set which contains the large element of i pairs and the small element from the other pairs of $R(S)$. It is easy to see that $S_{R_{\{k-i\}} \cup S_{\{i\}}} = \xi + (k-i)\rho + i\sigma = \xi + k\rho + i(\sigma - \rho)$, where ξ is the sum of all the small elements of R and S. □

LEMMA 4.7. $\max_i\{k_i\} + r - 1 \leq \log_2 \lambda$.

Proof. Using the fact that $d^i_j \geq 2d^i_{j-1}$ k_i times we get

$$\lambda \geq d^i_{k_i} \geq 2^{k_i} d^i_0 \geq 2^{k_i} d^1_0. \tag{18}$$

Since g_i divides g_{i-1} and is not equal to it we get $g_i \leq \frac{1}{2} g_{i-1}$, so

$$1 \leq g_r \leq \frac{1}{2^{r-1}} g_1 = \frac{1}{2^{r-1}} d^1_0. \tag{19}$$

Substituting (19) into (18) we get $\lambda \geq 2^{k_i} 2^{r-1}$, so $\log_2 \lambda \geq k_i + r - 1$. □

LEMMA 4.8. *Step 4 in the algorithm above is valid; i.e., before setting R^i_j (S^i_j), there are at least β^i_j pairs in $P_{\rho^i_j}$ $(P_{\sigma^i_j})$ that do not contain elements of X.*

Proof. Let $\prod_{i=1}^n y_i = Y$ and $2 \leq y_i < y$ for all $1 \leq i \leq n$, then

$$\frac{(y+1)}{\log y} > \frac{y_i}{\log y_i}$$

(the 1 is for the case where $y_i < e$). Therefore

$$y_i < (1+y)\log_y y_i,$$

so

$$\sum_i y_i < (y+1)\log_y Y.$$

Let $y_i = d^i_j/d^i_{j-1}$, $y = 2\lambda/|I|$, and $Y = d^i_{k_i}/d^i_0$. We have $2 \leq y_i < y$ by (15) and hence

$$\sum_{j=1}^{k_i} \frac{d^i_j}{d^i_{j-1}} \leq \left(1+\frac{2\lambda}{|I|}\right)\left(\frac{\log_2(d^i_{k_i}/d^i_0)}{\log_2(2\lambda/|I|)}\right) \leq \left(1+\frac{2\lambda}{|I|}\right)\left(\frac{\log_2\lambda}{\log_2(2\lambda/|I|)}\right).$$

We estimate the sum of all β's:

$$\begin{aligned}\sum_{i=1}^{r}\sum_{j=0}^{k_i}\beta^i_j &< 1+\tfrac{8\ell}{|I|}+1+\tfrac{4\lambda}{|I|}+r\left(2+\tfrac{2\lambda}{|I|}\right)+\textstyle\sum_{i=1}^r\sum_{j=0}^{k_i-1}\left(1+\tfrac{d^i_{j+1}}{d^i_j}\right)\\ &\leq 1+\tfrac{8\ell}{|I|}+1+\tfrac{4\lambda}{|I|}+r+r\max_i k_i+\left(1+\tfrac{2\lambda}{|I|}\right)r\left(1+\tfrac{\log_2\lambda}{\log_2\frac{2\lambda}{|I|}}\right).\end{aligned}$$

Since by (19) and (14), $r \leq 1+\log_2 d^1_0 \leq \log_2(2\lambda/|I|)$, it follows that

$$\sum_{i=1}^{r}\sum_{j=0}^{k_i}\beta^i_j < (2+\varepsilon)\lambda\frac{\log_2\lambda}{|I|}+\frac{8\ell}{|I|} < \frac{\min_{i\in I}p_i}{|I|\min_{i\in I}p_i}((2+\varepsilon)\lambda\log_2\lambda+8\ell).$$

Using (11) and the definition of M (and then (9)) we get that

$$\sum_{i=1}^{r}\sum_{j=0}^{k_i}\beta^i_j < \frac{\min_{i\in I}p_i}{\mu^2}(12.5\lambda\log_2^2\lambda+50\ell\log_2\lambda) \leq \frac{\min_{i\in I}p_i}{4},$$

and

$$4\sum_{i=1}^{r}\sum_{j=0}^{k_i}\beta^i_j \leq \min_{i\in I}p_i.$$

Therefore, in step 4 we always have β^i_j pairs for R^i_j and β^i_j pairs for S^i_j. □

Lemma 4.9 shows how to combine two arithmetic progressions in case they are long enough. It applies in both cases: when d_1 divides d_2 (Fig. 1) and when d_1 does not divide d_2 (Fig. 2). The first case is used by Lemma 4.10 for each $i = 1, \cdots, r$ to combine the arithmetic progressions with difference d^i_j, $j = 0, \cdots, k_i$ (obtained by Lemma 4.6) in the inner loop of step 3 of the algorithm. The resulting progression has difference d^i_0. The second case is used by Lemma 4.11 to combine the resulting progressions and derive a long enough arithmetic progression of difference g_r.

LEMMA 4.9. *Let X_1 and X_2 be two sets of integers such that*

$$X_1^* \supseteq \{s_1 + id_1\}_{i=1}^{n_1} \qquad \textit{and} \qquad X_2^* \supseteq \{s_2 + id_2\}_{i=2}^{n_2}$$

where

$$n_1 \geq \frac{d_2}{\gcd(d_1, d_2)} \qquad \textit{and} \qquad n_2 \geq \frac{d_1}{\gcd(d_1, d_2)}; \tag{20}$$

then

$$(X_1 \cup X_2)^* \supseteq \{s_3 + id_3\}_{i=1}^{n_3}$$

where

$$s_3 = s_1 + s_2 + \operatorname{lcm}(d_1, d_2), \qquad d_3 = \gcd(d_1, d_2),$$

and

$$n_3 = \frac{n_2 d_2 - \operatorname{lcm}(d_1, d_2) + d_1}{\gcd(d_1, d_2)}.$$

The sets forming the arithmetic progression can be described in constant time and any one of these sets can be built in time proportional to its cardinality.

Proof. We construct the desired arithmetic progression by showing:

$$\forall\, 1 \leq i \leq n_3, \qquad s_3 + id_3 = (s_1 + f_1(i)d_1) + (s_2 + f_2(i)d_2)\,. \tag{21}$$

Let e be the inverse of $d_1/\gcd(d_1, d_2)$ modulo $d_2/\gcd(d_1, d_2)$; then

$$f_1(i) \stackrel{\text{def}}{=} ie \left(\operatorname{mod} \frac{d_2}{\gcd(d_1, d_2)}\right), \qquad 0 < f_1(i) \leq \frac{d_2}{\gcd(d_1, d_2)}. \tag{22}$$

Note that we chose $f_1(i)$ so that $1 \leq f_1(i) \leq n_1$ and therefore $s_1 + f_1(i)d_1 \in X_1^*$.

$$f_2(i) \stackrel{\text{def}}{=} \frac{\operatorname{lcm}(d_1, d_2) + i \gcd(d_1, d_2) - f_1(i)d_1}{d_2}. \tag{23}$$

It follows from (22) that

$$f_1(i) \frac{d_1}{\gcd(d_1, d_2)} = i \left(\operatorname{mod} \frac{d_2}{\gcd(d_1, d_2)}\right)$$

and therefore $f_1(i)d_1 = i \gcd(d_1, d_2) \pmod{d_2}$. Hence the right-hand side of (23) is an integer and

$$(s_3 + id_3) - (s_1 + f_1(i)d_1 \ + \ s_2 + f_2(i)d_2) = 0$$

(by substituting (23), s_3 and d_3). Since $1 \leq i \leq n_3$, by the definition of n_3 we have

$$f_2(i) \leq \frac{\operatorname{lcm}(d_1, d_2) + n_3 \gcd(d_1, d_2) - 1 \cdot d_1}{d_2} \leq n_2$$

and

$$f_2(i) \geq \frac{\operatorname{lcm}(d_1, d_2) + 1 \cdot \gcd(d_1, d_2) - (d_2/\gcd(d_1, d_2))d_1}{d_2} > 0.$$

Hence $1 \leq f_2(i) \leq n_2$ and $s_2 + f_2(i)d_2 \in X_2^*$ and therefore $s_3 + id_3 \in (X_1 \cup X_2)^*$. Again, the time is constant for a description and linear for an elements list. □

LEMMA 4.10. *For $1 \leq i \leq r$ and $0 \leq j \leq k_i$ one can build an arithmetic progression with difference d_0^i and length $d_j^i(\beta_j^i - 1)$. The description of the sets whose sums give the arithmetic progression takes $O(\log \lambda)$ time and for each set B, listing its elements takes time proportional to its cardinality.*

Proof. The proof is by induction on j. The base hypothesis ($j = 0$) follows from Lemma 4.6 by taking $R = R_0^i$ and $S = S_0^i$. For the induction step we use Lemma 4.9 as follows: The first arithmetic progression is the combined arithmetic progression formed from the first j arithmetic progressions $(0, 1, \cdots, j-1)$. This progression follows from the induction hypothesis. It has $d_1 = d_0^i$ and $n_1 = (\beta_{j-1}^i - 1)d_{j-1}^i / d_0^i$. The second is the progression for d_j^i which follows from Lemma 4.6. It has $d_2 = d_j^i$ and $n_2 = \beta_j^i$. Obviously

$$n_2 = \beta_j^i \geq 1 = \frac{d_0^i}{d_0^i} = \frac{d_1}{\gcd(d_1, d_2)}.$$

So the first half of condition (20) of Lemma 4.9 holds. From the definition of β_{j-1}^i, it clearly follows that

$$n_1 = (\beta_{j-1}^i - 1)\frac{d_{j-1}^i}{d_0^i} \geq \frac{d_j^i}{d_0^i} = \frac{d_2}{\gcd(d_1, d_2)},$$

which proves the second half of condition (20). From Lemma 4.9 we get that the combined arithmetic progression has

$$\frac{\beta_j^i d_j^i - d_j^i + d_0^i}{d_0^i} \geq (\beta_j^i - 1)\frac{d_j^i}{d_0^i}$$

elements. As for the time complexity: we used Lemma 4.6 $k_i = O(\log \lambda)$ times; each takes $O(1)$ time. So we have $O(\log \lambda)$ time for a description and $O(k_i + |B|)$ for an elements list. The latter bound is $O(|B|)$, as $|B| = \Omega(k_i)$, since none of the sets R_j^i and S_j^i can be empty. □

LEMMA 4.11. *One can build an arithmetic progression with difference g_r and length $\frac{1}{2}d_{k_r}^r(\beta_{k_r}^r - 1)$. The description of the sets whose sums give the arithmetic progression takes $O(\log^2 \lambda)$ time and for each set B, listing its elements takes time proportional to its cardinality.*

Proof. We prove inductively on i that we can build an arithmetic progression with difference g_i and length $\frac{1}{2}d_{k_i}^i(\beta_{k_i}^i - 1)$ in $O(i \log \lambda)$ time. The base hypothesis ($i = 1$) trivially follows from Lemma 4.10. Suppose that the induction hypothesis holds for i; then for $i + 1$ we use Lemma 4.9. The first arithmetic progression is the combined arithmetic progression formed from the first i arithmetic progressions; this progression follows from the induction hypothesis. It has

$$d_1 = g_i \quad \text{and} \quad n_1 = \frac{d_{k_i}^i(\beta_{k_i}^i - 1)}{2g_i}.$$

The second is the progression from Lemma 4.10. It has

$$d_2 = d_0^{i+1} \quad \text{and} \quad n_2 = \frac{d_{k_{i+1}}^{i+1}(\beta_{k_{i+1}}^{i+1} - 1)}{d_0^{i+1}}.$$

By (15) we get that $d_{k_{i+1}}^{i+1} > d_0^{i+1}$. Using the definition of β_j^i for $j = k_i$, we get that

$$n_2 = \frac{d_{k_{i+1}}^{i+1}(\beta_{k_{i+1}}^{i+1} - 1)}{d_0^{i+1}} \geq \beta_{k_{i+1}}^{i+1} - 1 = 2\frac{\lambda}{|I|}. \tag{24}$$

On the other hand, from (14) it follows that

$$\frac{d_1}{\gcd(d_1, d_2)} = \frac{g_i}{g_{i+1}} \leq g_1 = d_0^1 \leq \frac{\lambda}{|I|}. \tag{25}$$

From (24) and (25) it follows that

$$n_2 \geq 2\frac{d_1}{\gcd(d_1, d_2)}. \tag{26}$$

From (16) and Lemma 4.3 (taking $a = d_0^i/g_i$ and $b = g_i$) it follows that

$$n_1 = \frac{d_{k_i}^i(\beta_{k_i}^i - 1)}{2g_i} \geq \frac{|I| d_0^i(\beta_{k_i}^i - 1)}{4g_i\alpha_{d_0^i}} \geq \frac{|I|(\beta_{k_i}^i - 1)}{4\alpha_{g_i}}.$$

On the other hand, from Lemma 4.5 it follows that

$$\frac{d_2}{\gcd(d_1, d_2)} = \frac{d_0^{i+1}}{g_{i+1}} \leq \frac{\lambda}{\alpha_{g_i} g_{i+1}}.$$

If $i < r - 1$, then $g_i > 1$ and therefore, substituting the definition of $\beta_{k_i}^i$, we get

$$\frac{d_2}{\gcd(d_1, d_2)} \leq \frac{\lambda}{2\alpha_{g_i}} \leq \frac{|I|(\beta_{k_i}^i - 1)}{4\alpha_{g_i}} \leq n_1.$$

Otherwise, $i = r - 1$ and

$$\frac{d_2}{\gcd(d_1, d_2)} \leq \frac{\lambda}{\alpha_{g_i}} \leq \frac{|I|(\beta_{k_i}^i - 1)}{4\alpha_{g_i}} \leq n_1.$$

Hence condition (20) holds and therefore we have

$$d_3 = \gcd(d_1, d_2) = g_{i+1},$$

and from (26) we get

$$\begin{aligned} n_3 = \frac{n_2 d_2 - \operatorname{lcm}(d_1, d_2) + d_1}{\gcd(d_1, d_2)} &\geq \frac{n_2 d_2 - \frac{d_1 d_2}{\gcd(d_1, d_2)}}{\gcd(d_1, d_2)} \geq \frac{n_2 d_2}{2\gcd(d_1, d_2)} \\ &= \frac{d_{k_{i+1}}^{i+1}(\beta_{k_{i+1}}^{i+1} - 1)}{2g_{i+1}}, \end{aligned}$$

proving the induction step. As for the time complexity: we used Lemma 4.10 $r = O(\log \lambda)$ times; each one of them takes $O(\log \lambda)$ time. So we have $O(\log^2 \lambda)$ time for a description and $O(r + |B|)$ for an elements list. Again, the latter bound is $O(|B|)$, as $|B| = \Omega(r)$, since none of the sets R_j^i and S_j^i can be empty. ◻

Completing the proof of Theorem 4.1. Applying (16) and the definition of $\beta^i_{k_i}$ to the result of Lemma 4.11, we get that the arithmetic progression generated has difference g_r and length

$$\frac{1}{2}d^r_{k_r}(\beta^r_{k_r}-1) \geq \frac{1}{2}\frac{d^r_0}{\alpha_{d^r_0}}\frac{|I|}{2}(\beta^r_{k_r}-1) > 2\ell\frac{d^r_0}{\alpha_{d^r_0}}.$$

Since $g_r | d^r_0$ by Lemma 4.3, $\alpha_{d^r_0} \leq (d^r_0/g_r)\alpha_{g_r}$, and since $\alpha_{g_r} = 1$, we get that the arithmetic progression has length of at least $2\ell g_r$ and thus at least 2ℓ elements.

To complete the proof, we analyze the time complexity.

1. Multiplying two polynomials of degree λ using FFT takes $O(\lambda \log \lambda)$ time (see for example [1]).
2. Sorting $\{p_i\}_{i=1}^{\lambda}$ takes $O(\lambda \log \lambda)$ time; choosing s and building $|I|$ takes $O(|I|)$, which is $O(\lambda)$ time.
3. There are two loops in this step: the outer loop where i goes from 1 to r, and the inner loop where j goes from 0 to k_i. We estimate each one of these two loops separately. We include the first iteration of the inner loop as part of the outer loop.
   (a) *Outer loop:* The outer loop is performed r times; each iteration takes $O(|I| + g_i)$ time for checking the while condition (g_i for initializing a vector of length g_i and constant time for updating some $|I(x, g_i)|$ for a single element in I). Choosing σ^i_0, ρ^i_0 takes $O(|I|)$ time. The first iteration of the inner loop takes $O(|I| + d^i_0)$ time. So the total time for the outer loop is $O(r(|I| + g_i + d^i_0))$.
   (b) *Inner loop:* The inner loop is performed no more than $r(\max_i k_i + 1) = O(r \log \lambda)$ times; each iteration takes $O(|I| + 2d^i_j/d^i_{j-1})$ time for checking the while condition and $O(|I|)$ time for choosing σ^i_j, ρ^i_j. From (15) it follows that

$$\frac{d^i_j}{d^i_{j-1}} \leq \frac{2\lambda}{|I|}.$$

   So the total time is $O(r \log \lambda(|I| + 2\lambda/|I|))$.

   From (14) it follows that

$$g_i \leq g_1 = d^1_0 \leq \frac{\lambda}{|I|},$$

   and using Lemma 4.5 we get $d^i_0 \leq \lambda$. From (19) and (14) we have

$$r \leq 1 + \log_2 d^1_0 \leq 1 + \log_2 \frac{\lambda}{|I|} = O\left(\frac{\lambda}{|I|}\right),$$

   and of course $r = O(\log \lambda)$. Thus the time complexity of this step is

$$\begin{aligned} &O\left(r|I| + rg_i + rd^i_0 + r\log\lambda|I| + r\log\lambda\tfrac{2\lambda}{|I|}\right) \\ =\; &O\left(\lambda + \log\lambda\tfrac{\lambda}{|I|} + \lambda\log\lambda + \lambda\log\lambda + \lambda\log\lambda\tfrac{r}{|I|}\right). \end{aligned}$$

   Using (12), the definition of M, and (9) we get that

$$|I| > \frac{M}{\mu} > \frac{\mu - 1}{2\log(e\lambda)} > 2\log_2\lambda > r.$$

   Hence the time complexity of this step is $O(\lambda \log \lambda)$.

4. Building R^i_j and S^i_j takes $O(\mu \log \mu + \sum_{i=1}^{r} k_i \mu) = O(\mu \log^2 \lambda)$.
5. The description of the arithmetic progression is generated in $O(\log^2 \lambda)$.

So the overall time complexity is $O(\lambda \log \lambda + \mu \log^2 \lambda)$.

5. A detailed description of the algorithm. In this section we combine all the theorems of previous sections, and prove a weak version of the main theorem of this article. First we introduce the following notations:

$$\mu \stackrel{\text{def}}{=} \left\lceil 10\ell^{0.5} \log_2 \ell \right\rceil . \tag{27}$$

$$L \stackrel{\text{def}}{=} \frac{1}{6} S_A + 2\ell^{1.5}. \tag{28}$$

THEOREM 5.1. *Let A be a set of m different numbers in the interval $(0, \ell]$ such that*

$$m > 100\ell^{0.5} \log_2 \ell; \tag{29}$$

then we can build in $O(\ell \log \ell + m \log^2 \ell)$ preprocessing time a structure which allows us to solve the subset-sum problem for any given integer N in the interval $(L, S_A - L)$. Solving means finding a subset $B \subseteq A$ such that $S_B \leq N$ and there is no subset $C \subseteq A$ such that $S_B < S_C \leq N$. The optimal subset B is built in $O(\log \ell)$ time per target number and only constant time for finding the optimal sum S_B .

ALGORITHM

1. The preprocessing stage consists of the following steps; the first two steps are sufficient for computing only the sum S_B.
   (a) Apply the algorithm of §3 to the input set A, using $t \stackrel{\text{def}}{=} \lceil \sqrt{\ell} \rceil$. Note that condition (29) guarantees that condition (1) of Theorem 3.4 holds.
   (b) Use dynamic programming modulo d_0 to compute $A^* \pmod{d_0}$. Use this to compute the function $f_{d_0} : [0, d_0) \to [0, d_0)$, which is defined as

   $$f_{d_0}(i) \stackrel{\text{def}}{=} \max\{j : 0 \leq j \leq i \text{ and } j \in A^* \pmod{d_0}\}.$$

   Note that $f_{d_0}(i)$ is properly defined for all i since $0 \in A^* \pmod{d_0}$.
   (c) In computing the set $A^* \pmod{d_0}$ keep a subset $C_i \subseteq A \setminus A(0, d_0)$ for each $i \in A^* \pmod{d_0}$ such that $S_{C_i} = i \pmod{d_0}$ and $S_{C_i} < \ell d_0$ (since by construction $|C_i| < d_0$).
   (d) We now turn to a new problem:

   $$A' = \frac{A(0, d_0)}{d_0}.$$

   From (2) and (1) we get that

   $$\begin{aligned} m' &> m - d_0 > \frac{39}{40} m, \\ L' &= L - \max_i \{S_{C_i}\} > L - \ell d_0, \end{aligned}$$

   and

   $$\ell' = \frac{\ell}{d_0} \leq \ell.$$

By (3) of Theorem 3.4, A' has a $d_0' = 1$. So we can apply again the algorithm of §3 to A' with the same t ((1) will certainly hold) and choose

$$A_1' \stackrel{\text{def}}{=} \Delta.$$

By (6), $S_{A_1'} < \frac{1}{3} S_{A'}$.

(e) Using (7) we have that

$$|A' \setminus A_1'| > \frac{2}{3} m' > \mu. \tag{30}$$

So we can choose $A_2' \stackrel{\text{def}}{=} (A' \setminus A_1')_{(\mu)}$ and apply the algorithm of §4 to A_2' with $\lambda = \ell$. Note that from the definition of μ it follows that condition (9) of Theorem 4.1 holds. Let B_i' $(0 \le i < 2\ell)$ be the subset of A_2', which satisfies $S_{B_i'} = s_0 + i g_r$.

(f) Using (14), (12), and the definition of M we get

$$g_r \le d_0^1 < \frac{\ell}{|I|} \le \frac{\ell \mu}{M} < \frac{4\ell \log(e\ell)}{\mu - 1}.$$

Using the definition of μ we get that

$$g_r < \frac{6\ell \log_2 \ell}{\mu} \le \frac{\ell^{0.5}}{2} \le t.$$

So we can use (5) and Lemma 3.3 to build, using dynamic programming, a sequence of sets $\{E_i'\}_{i=0}^{g_r - 1} \subseteq A_1'$ such that $S_{E_i'} = i \pmod{g_r}$ and $S_{E_i'} < \ell' g_r$.

(g) Define, for $0 \le i < \ell g_r$,

$$F_i' \stackrel{\text{def}}{=} E'_{i \pmod{g_r}} + B'_{\ell' + (i - S_{E'_{i \pmod{g_r}}})/g_r}.$$

It is easy to verify that the indices of B' and E' are valid and that $S_{F_i'} = s_0 + \ell' g_r + i$.

(h) Compute all the prefix sums of the set

$$A_3' \stackrel{\text{def}}{=} A' \setminus (A_1' \uplus A_2');$$

i.e., $G_i' \stackrel{\text{def}}{=} S_{A'_{3(i)}}$. By the choice of A_2', (30), and (7), it follows that

$$S_{A_2'} < \frac{\mu}{|A' \setminus A_1'|} S_{A' \setminus A_1'} < \frac{\mu}{\frac{2}{3}(m - d_0)} S_{A'} < \frac{S_{A'}}{6.2} \tag{31}$$

(since by (27) and (29), $m > 9.3\mu$). So by (4) of Theorem 3.4

$$S_{A_3'} = S_{A'} - S_{A_1'} - S_{A_2'} > \frac{S_A}{d_0} \left(1 - \frac{\ell}{\binom{m-t}{2}}\right) \left(1 - \frac{1}{3} - \frac{1}{6.2}\right) > \frac{S_A}{2d_0}.$$

2. Given a target number N we execute the following steps. For computing S_B only the first step suffices.

(a) Denote $n_0 \stackrel{\text{def}}{=} N \pmod{d_0}$. Compute S_B as follows:

$$S_B = d_0 \left\lfloor \frac{N}{d_0} \right\rfloor + f_{d_0}(n_0).$$

It is clear by taking numbers modulo d_0 that there is no C such that $S_B < S_C \leq N$. So if we can obtain S_B, then it is the right solution. In Lemma 5.2 we will prove that we can.

(b) Denote $\hat{S}_B \stackrel{\text{def}}{=} \min\{S_B, S_A - S_B\}$, $\hat{n}_0 \stackrel{\text{def}}{=} \hat{S}_B \pmod{d_0}$ and

$$N' \stackrel{\text{def}}{=} \frac{\hat{S}_B - S_{C_{f_{d_0}(\hat{n}_0)}}}{d_0} - s_0 - \ell' g_r.$$

(c) Using binary search on the set of prefix sums of A'_3, find

$$n_1 \stackrel{\text{def}}{=} \max\{i : S_{G'_i} \leq N'\} \quad \text{and} \quad n_2 \stackrel{\text{def}}{=} N' - S_{G'_{n_1}}.$$

The desired subset is

$$B_? \stackrel{\text{def}}{=} C_{f_{d_0}(\hat{n}_0)} \uplus d_0 G'_{n_1} \uplus d_0 F'_{n_2}.$$

LEMMA 5.2. *The algorithm above computes the correct set B.*

Proof. The indices n_1 and n_2 are valid: $s_0 < S_{A'_2}$ and from (31) it follows that

$$N' > \frac{L - \ell d_0}{d_0} - \frac{S_A}{6.2 d_0} - \frac{\ell g_r}{d_0} > \frac{L - 2lt - \frac{1}{6} S_A}{d_0} \geq 0.$$

And clearly $N' \leq S_A / 2d_0$, therefore $S_{A'_3} > N'$ and hence n_1 exists. $n_2 \geq 0$ is trivial and $n_2 < \ell$ follows from the fact that all the elements in A_3 are bounded by ℓ. The sum of $B_?$ is:

$$\begin{aligned} S_{B_?} &= S_{C_{f_{d_0}(\hat{n}_0)}} + d_0 \left(S_{G'_{n_1}} + s_0 + \ell' g_r + (N' - S_{G'_{n_1}}) \right) \\ &= S_{C_{f_{d_0}(\hat{n}_0)}} + d_0 \left(s_0 + \ell' g_r + N' \right). \end{aligned}$$

Substituting N' yields $S_{B?} = \hat{S}_B$ which is S_B or $S_A - S_B$; in the second case we complement the solution ($B = A \setminus B_?$). □

The time complexity.

1. The preprocessing time is
   (a) The algorithm of §3 takes $O(m + t^2)$ time.
   (b) Solving the problem modulo d_0 takes $O(d_0^2) = O(t^2)$ time.
   (c) Keeping the subsets C along the way does not cost any additional time.
   (d) The second application of the algorithm of §3 is even faster than the first one.
   (e) Sorting A takes $O(m \log m)$ time and the Algorithm of §4 takes

   $$O(\ell \log \ell + m \log^2 \ell)$$

   time.
   (f) This dynamic programming takes $O(m + g_r^2) = O(m + t^2)$ time.
   (g) Building any B_i and F'_i might take $O(m)$ time, so we do not actually build it; instead we keep the necessary index computations only.

(h) Building all the G_i''s takes $O(m)$ time.

Thus, the time complexity of the whole preprocessing is

$$O\left(m\log^2\ell + t^2 + \ell\log\ell\right).$$

By the definition of t the complexity is

$$O(m\log^2\ell + \ell\log\ell).$$

2. The per-target number part of the algorithm takes the following:
   (a) Finding the solution S_B takes only constant time.
   (b) Computing n_1 and n_2 takes constant time as well.
   (c) The binary search takes $O(\log m)$ time; all the other indices can be computed in constant time, a characterization can be obtained in logarithmic time, and a full element list can be written in $O(|B|) = O(m)$ time.

6. Improvement of the algorithm. In this section we improve the algorithm described in the previous section. We get a faster algorithm (even linear in some cases) which works on a larger interval. The bottleneck of the time complexity is $O(\lambda\log\lambda)$ where $\lambda = \ell$, so the main idea of the improvement is to take λ to be less than ℓ. The smaller interval size could be obtained for the previous algorithm using better estimates.

First we introduce the following choice of parameters:

$$t \stackrel{\text{def}}{=} \left\lceil \frac{103\ell}{m}\log_2\ell \right\rceil, \quad \mu \stackrel{\text{def}}{=} \left\lceil 15\ell^{0.5}\log_2\ell \right\rceil, \quad \text{and} \quad L \stackrel{\text{def}}{=} \frac{100 S_A \ell^{0.5}\log_2\ell}{m}. \tag{32}$$

THEOREM 6.1. *Let A be a set of m different numbers in the interval $(0,\ell]$ such that*

$$m > 1000\ell^{0.5}\log_2\ell; \tag{33}$$

then we can build in $O\left(m + ((\ell/m)\log\ell)^2 + (S_A/m^2)\ell^{0.5}\log^2\ell\right)$ preproccesing time a structure which allows us to solve the subset-sum problem for any given integer N in the interval $(L, S_A - L)$. Solving means finding a subset $B \subseteq A$ such that $S_B \le N$ and there is no subset $C \subseteq A$ such that $S_B < S_C \le N$. An optimal subset B is built in $O(\log\ell)$ time per target number and is listed in time $O(|B|)$. For finding the optimal sum S_B only, the preprocessing time is $O\left(m + ((\ell/m)\log\ell)^2\right)$ and only constant time is needed per target number.

ALGORITHM

1. The preprocessing stage consists of the following steps; the first two steps are sufficient for computing only the sum S_B.
   (a) Apply the algorithm of §3 to the input set A, using t as defined before. Condition (1) of Theorem 3.4 clearly holds.
   (b) Use dynamic programming modulo d_0 to compute $A^*\ (\text{mod } d_0)$. Use this to compute the function $f_{d_0} : [0, d_0) \to [0, d_0)$, which is defined as

   $$f_{d_0}(i) \stackrel{\text{def}}{=} \max\{j : 0 \le j \le i \text{ and } j \in A^*\ (\text{mod } d_0)\}.$$

   Note that $f_{d_0}(i)$ is properly defined for all i since $0 \in A^*\ (\text{mod } d_0)$.
   (c) In computing the set $A^*\ (\text{mod } d_0)$ keep a subset $C_i \subseteq A \setminus A(0, d_0)$ for each $i \in A^*\ (\text{mod } d_0)$ such that $S_{C_i} = i\ (\text{mod } d_0)$ and $S_{C_i} < \ell d_0$.

(d) We now turn to a new problem:

$$A' = \frac{A(0, d_0)}{d_0}.$$

From (2) and (1) we get that

$$m' > m - d_0 > \frac{39}{40}m,$$

and

$$\ell' = \frac{\ell}{d_0} \leq \ell.$$

By (3) of Theorem 3.4, A' has a $d'_0 = 1$. So we can apply again the algorithm of §3 to A' with the same t ((1) will certainly hold) and choose $A'_1 \overset{\text{def}}{=} \Delta$. By (6), $S_{A'_1} < \frac{1}{3} S_{A'}$.

(e) This step is the core of the improvement of the algorithm. Using (7) we have that

$$|A' \setminus A'_1| > \frac{2}{3}m' > \frac{1}{2}m. \tag{34}$$

Let

$$\lambda \overset{\text{def}}{=} \left\lceil 64\ell^{0.5} \log_2 \ell \frac{4S_{A'}}{m^2} \right\rceil. \tag{35}$$

By the density assumption and since $S_{A'} < \ell m$, we have that $\lambda < \ell$. Lemma 6.2 below enables us to choose a subset $A'_2 \subseteq A' \setminus A'_1$, which contains μ elements, each one of which is less than $4S_{A'}/m$ and lies in a subinterval of length λ. Apply the algorithm of §4 to A'_2. Note that from the definition of μ it follows that condition (9) of Theorem 4.1 holds. Let B'_i $(0 \leq i < 2\ell)$ be the subset of A'_2 which satisfies $S_{B'_i} = s_0 + ig_r$.

(f) Using (14), (12), and the definition of M we get

$$g_r \leq d_0^1 < \frac{\lambda}{|I|} \leq \frac{\lambda\mu}{M} < \frac{4\lambda \log(e\ell)}{\mu - 1}.$$

Using the definition of μ we get that

$$g_r < \frac{6\lambda \log_2 \ell}{\mu} \leq \frac{103 S_{A'}}{m^2} \log_2 \ell < \frac{103\ell}{m} \log_2 \ell \leq t. \tag{36}$$

So we can use (5) and Lemma 3.3 to build, using dynamic programming, a sequence of sets $\{E'_i\}_{i=0}^{g_r - 1} \subseteq A'_1$ such that $S_{E'_i} = i \pmod{g_r}$ and $S_{E'_i} < \ell' g_r$.

(g) Define, for $0 \leq i < \ell g_r$,

$$F'_i \overset{\text{def}}{=} E'_{i \pmod{g_r}} + B'_{\ell' + (i - S_{E'_{i \pmod{g_r}}})/g_r}.$$

It is easy to verify that the indices of B' and E' are valid and that $S_{F'_i} = s_0 + \ell' g_r + i$.

(h) Compute all the prefix sums of the set

$$A'_3 \stackrel{\text{def}}{=} A' \setminus (A'_1 \cup A'_2);$$

i.e., $G'_i \stackrel{\text{def}}{=} S_{A'_{3(i)}}$. A'_2 contains μ elements, each one of which is less than $4S_{A'}/m$ so

$$S_{A'_2} < \mu \frac{4S_{A'}}{m} < \frac{64 S_{A'} \ell^{0.5} \log_2 \ell}{m} < \frac{S_{A'}}{10}. \tag{37}$$

So by (4) of Theorem 3.4

$$S_{A'_3} = S_{A'} - S_{A'_1} - S_{A'_2} > \frac{S_A}{d_0}\left(1 - \frac{\ell}{\binom{m-t}{2}}\right)\left(1 - \frac{1}{3} - \frac{1}{10}\right) > \frac{S_A}{2d_0}.$$

2. Given a target number N we execute the following steps. For computing S_B only the first step suffices.

(a) Denote $n_0 \stackrel{\text{def}}{=} N \pmod{d_0}$. Compute S_B as follows:

$$S_B = d_0 \left\lfloor \frac{N}{d_0} \right\rfloor + f_{d_0}(n_0).$$

It is clear by taking numbers modulo d_0 that there is no C such that $S_B < S_C \le N$. So if we can obtain S_B, then it is the right solution. In Lemma 6.3 we will prove that we can.

(b) Denote $\hat{S}_B \stackrel{\text{def}}{=} \min\{S_B, S_A - S_B\}$, $\hat{n}_0 \stackrel{\text{def}}{=} \hat{S}_B \pmod{d_0}$ and

$$N' \stackrel{\text{def}}{=} \frac{\hat{S}_B - S_{C_{f_{d_0}(\hat{n}_0)}}}{d_0} - s_0 - \ell' g_r.$$

(c) Using binary search on the set of prefix sums of A'_3, find

$$n_1 \stackrel{\text{def}}{=} \max\{i : S_{G'_i} \le N'\} \quad \text{and} \quad n_2 \stackrel{\text{def}}{=} N' - S_{G'_{n_1}}.$$

The desired subset is

$$B_? \stackrel{\text{def}}{=} C_{f_{d_0}(\hat{n}_0)} \cup d_0 G'_{n_1} \cup d_0 F'_{n_2}.$$

LEMMA 6.2. *There exists a subset $A'_2 \subseteq (A' \setminus A'_1)$ which has μ elements, each one of which is less than $4S_{A'}/m$ and is contained in an interval of length λ where λ is defined in* (35). *This subset can be found in time $O(m)$.*

Proof. Denote the median of the set $A' \setminus A'_1$ by H. Since

$$|A' \setminus A'_1| \ge \frac{2}{3} m' > \frac{2}{3}\frac{39}{40} m = \frac{13}{20} m,$$

at least $m/4$ elements of this set are larger than the median:

$$S_{A' \setminus A'_1} > H \frac{m}{4},$$

which implies that

$$H < \frac{4 S_{A' \setminus A'_1}}{m} \le \frac{4 S_{A'}}{m}.$$

So there are at least $m/4$ elements in $A' \setminus A'_1$ which are smaller than $4S_{A'}/m$. Divide the interval $(0, 4S_{A'}/m]$ into $\lceil (4S_{A'}/m\lambda) \rceil$ subintervals of length less than or equal to λ each and choose the subinterval with the largest number of elements. It contains at least

$$\frac{m}{4} / \left\lceil \frac{4S_{A'}}{m} / \lambda \right\rceil > \mu$$

elements. Arbitrarily enlarge the subinterval to the length λ and take only μ elements from it. A'_2 can be easily computed in linear time. □

LEMMA 6.3. *The algorithm above computes the correct set B.*

Proof. From (2) it follows that

$$S_A > \binom{m - d_0}{2} d_0 > \frac{m^2 d_0}{3}.$$

From (37) and the fact that $S_{A'} \le S_A/d_0$ we have

$$s_0 \le S_{A'_2} < \frac{64 S_A \ell^{0.5} \log_2 \ell}{m d_0}.$$

We obviously have $d_0 \le 3S_A/m^2$ and by (36)

$$g_r \le \frac{103 S_A \log_2 \ell}{d_0 m^2}.$$

It follows from these inequalities, the definition of L, and the density condition that

$$N' > \frac{L - \ell d_0}{d_0} - s_0 - \frac{\ell g_r}{d_0} > \left(L - l\frac{3S_A}{m^2} - \frac{64 S_A \ell^{0.5} \log_2 \ell}{m} - \frac{103 S_A \ell}{m^2} \log_2 \ell \right) / d_0 \ge 0.$$

Clearly $N' \le S_A/(2d_0)$ and therefore $S_{A'_3} > N'$ and hence n_1 exists. $n_2 \ge 0$ is trivial and $n_2 < \ell$ follows from the fact that all the elements in A_3 are bounded by ℓ. So the indices n_1 and n_2 are valid. The sum of $B_?$ is:

$$\begin{aligned} S_{B_?} = \quad & S_{C_{f_{d_0}(\hat{n}_0)}} + d_0 \left(S_{G'_{n_1}} + s_0 + \ell g_r + (N' - S_{G'_{n_1}}) \right) \\ = \quad & S_{C_{f_{d_0}(\hat{n}_0)}} + d_0 \left(s_0 + \ell g_r + N' \right). \end{aligned}$$

Computing this expression yields $S_{B?} = \hat{S}_B$ which is S_B or $S_A - S_B$; in the second case we complement the solution. □

The time complexity.

1. The preprocessing time is
   (a) The algorithm of §3 takes $O(m + t^2)$ time.
   (b) Solving the problem modulo d_0 takes $O(d_0^2) = O(t^2)$ time.
   (c) Keeping the subsets C along the way does not cost any additional time.
   (d) The second application of the algorithm of §3 is even faster than the first one.
   (e) Finding A'_2 takes $O(m)$ time with the algorithm described in §4 takes $O(\lambda \log \lambda + \mu \log^2 \lambda)$ time.
   (f) Dynamic programming takes $O(m + g_r^2) = O(m + t^2)$ time.
   (g) Building any B_i and F'_i might take $O(m)$ time, so we do not actually build it; instead we keep the necessary index computations only.

(h) Building all the G_i''s takes $O(m)$ time.
Thus, the total complexity of the preprocessing is

$$O\left(m + t^2 + \lambda \log \lambda + \mu \log^2 \lambda\right).$$

By the definition of t and λ the complexity is

$$O\left(m + \left(\frac{\ell}{m}\log \ell\right)^2 + \frac{S_A}{m^2}\ell^{0.5}\log^2 \ell\right),$$

since the last term is bounded by $O(\ell^{0.5}\log^3 \ell)$, which is dominated by one of the first two terms because of the density condition.

2. The per-target number part of the algorithm takes
   (a) Finding the solution S_B takes only constant time.
   (b) Computing n_1 and n_2 takes constant time as well.
   (c) The binary search takes $O(\log m)$ time; all the other indices can be computed in constant time, a characterization can be obtained in logarithmic time, and a full element list can be written in $O(|B|) = O(m)$ time.

Remarks. Note that L is always smaller than S_A ($L < S_A/10$). This was not true in the previous section and in the previous known results. For example, when $m = o(\ell^{3/4})$ and $L = \ell^{1.5}$, S_A can be $O(m^2) = o(L)$ (i.e., when there are very few elements close to ℓ) and the interval $(L, S_A - L)$ in Theorem 5.1 is empty.

The time complexity of our algorithm is bounded above by

$$O\left(m + \frac{\ell^{1.5}}{m}\log^2 \ell\right).$$

This expression can be $O(m)$ when $m = \Omega(\ell^{3/4}\log^{1.5}\ell)$, but can also hold for sparser instances for which S_A is relatively small:

$$m = \Omega((\ell \log \ell)^{2/3}) \quad \text{and} \quad S_A = O\left(\frac{m^3}{\ell^{0.5}\log^2 \ell}\right).$$

If we are only interested in the optimal sum, the time complexity is bounded by $O(m + \left((\ell/m)\log \ell)^2\right)$. This expression is $O(m)$ when $m = \Omega((\ell \log \ell)^{2/3})$.

7. Limitations of our method. In this section we investigate the limits of our techniques. First we show a lower bound on the density below which the method does not work and then we show some problems in which density does not help.

A lower bound on the density. In our algorithm we proved that for $m > c\,\ell^{1/2+\varepsilon}$, A has a special structure. Namely, a large neighborhood of the middle sum ($S_A/2$) is a collection of arithmetic progressions with the same small difference:

$$A^* \cap (L, S_A - L) = \{a \in (L, S_A - L) \mid a \in A^* \pmod{d}\} \quad \text{where } d < m.$$

We call A with such a structure a *modulo-d-dependent* set. The previous works mentioned before, [9], [17], and [2] proved that A has this structure. We improved these results but we are still quite far from Freiman's conjecture that the density $m > \ell^\varepsilon$ suffice. Here we give a counterexample which proves that with this structure we cannot have density below $\ell^{1/2}$. We give an example in which for $m > \ell^{1/2}$, A^* does not have the structure described above.

EXAMPLE 7.1. *Let k be any even integer and let $A = \{1, 2, \cdots, k, \ell\}$, where $\ell = k(k+1)/2 + 2$. Then $\frac{1}{2}S_A$ is the only sum which cannot be achieved as a subset sum.*

Proof. $S_A = 2\ell - 2$, so $N = \frac{1}{2}S_A = \ell - 1$ cannot be achieved, since ℓ itself cannot be in the subset, because $\ell > N$ and all the other elements do not suffice since their sum is $\ell - 2 < N$. On the other hand, all the other values can be achieved: for $0 \leq N < \ell - 1$, we can easily find a subset of the first k elements whose sum is N, and using the ℓ element we can build sets for the other sums. In this example $m = k + 1 > \ell^{1/2}$. □

COROLLARY 7.2. *In the example above, A is not modulo-d-dependent for any d and L such that $d + L < \frac{1}{2}S_A$.*

Proof. $\frac{1}{2}S_A \notin A^*$ and this is exactly the middle value; therefore we should have some other value ruled out from A^* because $\frac{1}{2}S_A - d \in (L, S_A - L)$. But there is no other value which cannot be achieved — a contradiction. □

The example above shows that for density smaller than $\ell^{1/2}$, A^* does not have the same structure which we used in the algorithm. Thus the example above establishes a tight bound for the density (up to a polylog factor). However, it may be the case that for less dense instances, another, perhaps more complicated, structure exists.

Strong NP-completeness of dense problems. We solved instances of a dense NP-complete problem, but not of a strongly NP-complete one. The following theorems show that our techniques cannot be applied to obtain a pseudo-polynomial-time for some strongly NP-complete problems. We consider the bin packing problem and the three partition problem. Both are strongly NP-complete [14]. For each one, we describe a pseudo-polynomial-time reduction of a general instance to a very dense instance of the same problem; establishing that the very dense version of the corresponding problem is strongly NP-complete.

As a first example we consider the bin packing problem.

DEFINITION. An instance of the bin packing problem (BPP) is a triplet $\langle A, N, n\rangle$ where A is a set of integers and n, N are integers. A solution is n disjoint sets of integers $\{B_i\}_{i=1}^n$ such that $A = \uplus_{i=1}^n B_i$ and for all $1 \leq i \leq n$, $S_{B_i} \leq N$.

DEFINITION. An instance of the BPP is said to be *very dense* if $|A| > \frac{1}{2}\max A$.

THEOREM 7.3. *The* BPP *is strongly* NP*-complete even if restricted to very dense instances.*

Proof. Given an instance $I = \langle A, N, n\rangle$ of the BPP we generate a very dense instance $I' = \langle A', N', n'\rangle$ of the BPP as follows. Let

$$n' = \left\lfloor \frac{nN}{2} \right\rfloor, \qquad N' = 3nN,$$

and

$$A' = \{na \mid a \in A\} \cup \{nN + i\}_{i=n}^{n'-1} \cup \{2nN - i\}_{i=0}^{n'-1} \cup \{i\}_{i=1}^{n-1}.$$

Note that if A has no repetitions, so does A'. (We assume that $\max A \leq N$, otherwise I has no solution.) I' is very dense because it has $|A| + 2n' - 1$ elements in an interval of length $2nN$, so $|A'| > \frac{1}{2}\max A$.

We show that I is solvable if and only if I' is. If $\{B_i\}_{i=1}^n$ is a solution for I, i.e., $S_{B_i} \leq N$, then

$$B'_{i+1} = \begin{cases} B_1 \cup \{2nN\} & i = 0, \\ B_{i+1} \cup \{2nN - i, i\} & 1 \leq i < n, \\ \{2nN - i, nN + i\} & n \leq i < n' \end{cases}$$

is a solution for I'.

For the other implication, assume that $\{B'_i\}_{i=1}^{n'}$ is a solution for I'. For $0 \le i < n'$,

$$2nN - i > \frac{3nN}{2} = \frac{N'}{2},$$

and we must have a separate bin for each element in $\{2nN - i\}_{i=0}^{n'-1}$, which exhausts all our bins. Now we argue inductively that for each i, $n \le i < n'$, $nN + i$ should be packed together with its $\{2nN - i\}$ mate. The basis ($k = n - 1$) is obvious. Assume that this holds for $i > k$. Then for $nN + k$ there is just one bin with enough space left in it, which proves the induction step. So we are left with only n bins, say B''_i, $1 \le i < n$; bin B''_i contains the element $2nN - i$. The remaining elements in B''_i sum up to at most $nN + i$. All the remaining elements have sizes which are either a multiple of n or less than n, so the extra space modulo n left is useless for the large elements and exactly sufficient for the small ones. Consequently,

$$B_{i+1} \stackrel{\text{def}}{=} \left\{ \left\lfloor \frac{a'}{n} \right\rfloor : a' \in B''_i \right\}, \qquad 0 < i \le n,$$

solves I. □

COROLLARY 7.4. *The problem of partitioning into n equal parts is strongly* NP-*complete even if restricted to very dense instances.*

Proof. The same transformation may be used since it preserves the exactness of the packing. □

As a second example, we consider the 3-partition problem.

DEFINITION. An instance of the 3-partition problem (3PP) is a triplet $\langle A, m, \ell \rangle$, where A is a set of $3m$ elements in the interval $[1, \ell]$. The problem is whether there exists a partition of the set A into m triplets with the same sum.

DEFINITION. A 3PP instance is called very dense if $m > \frac{1}{8}\ell$.

THEOREM 7.5. 3PP *is strongly* NP-*complete even if restricted to very dense instances.*

Proof. The transformation from any input instance $I = \langle A, m, \ell \rangle$ into a very dense one $I' = \langle A', m', \ell' \rangle$ is as follows. Let

$$\ell' = 2\ell, \qquad m' = m + \left\lfloor \frac{1}{4}\ell \right\rfloor,$$

and

$$A' = \{a + \ell \mid a \in A\} \cup \left\{ 2i - 1, 2i, \frac{S_A}{m} + 3\ell - 4i + 1 \right\}_{i=1}^{\lfloor \frac{1}{4}\ell \rfloor}.$$

I' is clearly very dense.

If I has a solution, so does I': partition the first part of A' into triplets as A was partitioned and add new triples of the form $\{2i-1, 2i, S_A/m + 3\ell - 4i + 1\}$. Conversely, assume I' has a solution. Note that the sum of a triple is $S_{A'}/m' = S_A/m + 3\ell$ and therefore for $1 \le i \le \lfloor \ell/4 \rfloor$,

$$\frac{S_A}{m} + 3\ell - 4i + 1 > \max\left\{ \frac{1}{2}\frac{S_{A'}}{m'}, \frac{S_{A'}}{m} - \ell \right\}.$$

Consequently a triple with $S_A/m + 3\ell - 4i + 1$ can contain neither an element of the same form nor an element of the form $a + \ell$ (since there is not enough room left). So we need to fill this triple up with $2i - 1$ and $2i$, which are exactly sufficient for this purpose. We can read the solution for I from the remaining triples. □

8. Conclusion. We described a new algorithm for solving the subset-sum problem. It is always two orders of magnitude faster than the algorithm using dynamic programming. It is at least one order of magnitude, and for low densities two orders of magnitude, faster than the best algorithm using analytic number theory. In addition, the new algorithm uses only elementary number-theoretic facts and yields elementary proofs of theorems previously proved using analytic number theory.

The constants in the algorithm and the theorem are quite large. Moreover, we require that there is an ℓ_0 such that $\ell > \ell_0$. A careful proof yields much smaller constants but is much more tedious. Chaimovich [6] has carefully computed the constants in a previous, less efficient, algorithm which used analytical number-theoretic results. He has found that the algorithm works for $\ell_0 \geq 49$. Furthermore, he programmed it and discovered that this algorithm works even when the conditions (on the density and the interval of target numbers) are relaxed considerably. We believe that the true conditions for our algorithm are much weaker and that large constants are not needed.

Our results complement the results of Lagarias and Odlyzko [16] that considered sparse subset-sum problems. For these cases they designed a polynomial-time algorithm which finds the solution almost always. For dense subset-sum problems, the dynamic programming approach yields a polynomial-time algorithm. Our substantial speedup increases the size of the domain of solvable dense subset-sum problems considerably. Thus, one should be very careful before using instances of the subset-sum problem in cryptography (see [15]).

Other NP-complete problems should be examined. One should try to find an efficient algorithm for the dense case or prove that the problem remains NP-complete even when it is very dense. It is an open question whether there are strong NP-complete problems with a polynomial-time solution for the dense case.

One possibility for exploiting our result is to develop *density preserving* reductions of other NP-complete problems to the subset-sum problem. Recently Chaimovich [5] reduced several problems to the subset-sum problem using dense reductions obtaining improved algorithms for these problems. They include the problem of partitioning into k equal parts (for a fixed k) and certain scheduling problems.

One may want to try to use number theory (analytic or elementary) to solve other integer programming problems, possibly by analyzing their mathematical structure. After all, the inputs to these problems consist of numbers.

Appendix: A short history. There were two related developments: one was proving characterization theorems and the other was the development of algorithms. The earliest characterization theorems of discrete optimization problems were reported by Berstein and Freiman [3] and Freiman [10]. The first characterization theorem for the subset-sum problem showing that an interval near the middle sum is contained in A^* was proved by Erdös and Freiman [9]. It assumed very dense inputs ($m > \ell/3$) and a very small interval. Alon and Freiman [2] improved the density ($m > \ell^{2/3+\epsilon}$) and Lipkin [17] improved the interval size but needed higher density ($m > \ell^{4/5+\epsilon}$). All these works used analytic number theory. An earlier version of our work, using elementary means, yielded a characterization that matched both the density of [2] and the interval size of [17]. Then Freiman [13] and, independently, Sarközy [18] improved the density to $m > \ell^{.5+\epsilon}$. Freiman uses analytic number theory. Sarközy's proof does not, but it uses some algebra and is not constructive and therefore it is not useful for algorithm design. Our current paper matches this density using elementary means and proves it is the best possible.

The first algorithm using a similar approach is due to Freiman [11] (see also [12]). He used the result of [2] to obtain a linear-time ($O(m)$) algorithm for the simpler problem of computing S_B, the best sum, not the best set. The computation of B requires a large interval, which the theorem in [2] did not give. So the first algorithm for computing B used [17] and therefore needed higher density. Its time was far from linear. It was considerably improved by an earlier version of our paper. As a result of Freiman's more recent characterization [13], the best algorithm for computing B using analytical number theory [7] applies to density $m > \ell^{.5+\epsilon}$ but runs in time $O(\ell^2 \log \ell)$.

The earlier version of our paper obtained similar results to the ones reported here, except that it applied to density $m > \ell^{2/3+\epsilon}$ only. In this newer version we improve the density to $m > \ell^{.5+\epsilon}$. The time complexity of our algorithm considerably improves both the algorithm using dynamic programming and the best algorithm obtained using analytic number theory.

In case we are interested in computing S_B only, our algorithm has a better time bound. In particular, like Freiman's original algorithm, it is linear ($O(m)$) for densities $m > \ell^{2/3+\epsilon}$. Furthermore, if we solve this simpler problem for many target numbers, the additional time per target number is constant.

Acknowledgments. We thank M. Chaimovich and G. Frieman for their comments and suggestions.

REFERENCES

[1] A. V. AHO, J. HOPCROFT, AND J. B. ULLMAN, *The design and analysis of computer algorithms*, Addison–Wesley, Reading, MA, 1974, pp. 201–206.

[2] N. ALON AND G. A. FREIMAN, *On sums of subsets of a set of integers*, Combinatorica, 8 (1988), pp. 305–314.

[3] A. A. BERSTEIN AND G. A. FREIMAN, *Analytical methods of discrete optimization*, ZEMJ, (1979), pp. 89–105.

[4] M. CHAIMOVICH, *An efficient algorithm for the subset-sum problem*, manuscript, 1988.

[5] ———, *Fast exact and approximate algorithms for k-partition and scheduling independent tasks*, in Proc. French–Israeli Conference on Combinatorics and Algorithms, November 1988.

[6] ———, *Subset sum problems with different summands, computation* (*note*), Discrete Appl. Math., 27 (1990), pp. 277–282.

[7] M. CHAIMOVICH, G. A. FREIMAN, AND Z. GALIL, *Solving dense subset-sum problems by using analytic number theory*, J. Complexity, 5 (1989), pp. 271–282.

[8] G. B. DANTZIG, *Discrete-variable extremum problems*, Oper. Res., 5 (1957), pp. 266–277.

[9] P. ERDÖS AND G. A. FREIMAN, *On two additive problems*, J. Number Theory, 34 (1990), pp. 1–12.

[10] G. A. FREIMAN, *An analytical method of analysis of linear Boolean equations*, Ann. New York Acad. Sci., 337 (1980), pp. 97–102.

[11] ———, *On extremal additive problems of Paul Erdös*, in Proc. Canberra Conference on Combinatorics, August 1987; ARS Combin., 26B (1988), pp. 93–114.

[12] ———, *Subset-sum problem with different summands*, Congr. Numer., 70 (1990), pp. 207–215.

[13] ———, *A new analytic result on the subset-sum problem*, in Proc. French–Israeli Conference on Combinatorics and Algorithms, November 1988.

[14] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

[15] R. IMPAGLIAZZO AND M. NAOR, *Efficient cryptographic schemes provably as secure as subset sum*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, October 1989, pp. 236–241.

[16] J. LAGARIAS AND A. ODLYZKO, *Solving low-density subset sums*, in Proc. 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, October 1983, pp. 1–10.

[17] E. LIPKIN, *On representation of r-powers by subset sums*, Acta Arithmetica, L11 (1989), pp. 353–366.

[18] A. SARKÖZY, *Finite addition theorems*, II, J. Number Theory, to appear.